Titre: Title:	Sécurité des agents mobiles : protection des agents mobiles par un clone de référence
Auteur: Author:	Lotfi Benachenhou
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Benachenhou, L. (2003). Sécurité des agents mobiles : protection des agents mobiles par un clone de référence [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7022/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7022/
Directeurs de recherche: Advisors:	
Programme: Program:	Unspecified

UNIVERSITÉ DE MONTRÉAL

SÉCURITÉ DES AGENTS MOBILES : PROTECTION DES AGENTS MOBILES PAR UN CLONE DE RÉFÉRENCE

LOTFI BENACHENHOU
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
Avril 2003

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

SÉCURITÉ DES AGENTS MOBILES : PROTECTION DES AGENTS MOBILES PAR UN CLONE DE RÉFÉRENCE

présenté par: BENACHENHOU Lotfi

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen composé de:

M. QUINTERO Alejandro, Ph.D., Président

M. PIERRE Samuel, Ph.D., directeur de recherche

Mme. BOUCHENEB Hanifa, Ph.D., membre

Aux victimes du World Trade Center, 11 Septembre 2001,

Aux victimes de la guerre en Afghanistan, Octobre 2001,

Aux victimes des la catastrophe de Bab El Oued (Alger), 10 Novembre 2001,

Aux victimes de la guerre en Irak, Mars/Avril 2003,

Aux victimes du terrorisme en Algérie, depuis 1990 jusqu'à ...ce jour!!

REMERCIEMENTS

Je désire remercier mon directeur de recherche, le professeur Samuel Pierre, pour sa patience, ses conseils et ses commentaires.

Je désire également remercier tous les membres du Laboratoire de Recherche en Réseautique et Informatique Mobile (LARIM), et en particulier Abdelmorhit El Rhazi, Ali Chamam, Dougoukolo Konaré, Khaled Laouamri, et Younes Zaki, pour leur aide et leurs critiques.

Je tiens à remercier aussi M. Francis Gagnon qui a mis à ma disposition la salle de tests, ainsi que pour sa disponibilité.

En outre, je désire remercier toute ma famille, et en particulier mes parents, qui m'ont soutenu tout au long de mes études, et sans qui je n'aurais jamais atteint ce niveau d'études. Je remercie finalement mes amis pour leurs encouragements et leur soutien.

RÉSUMÉ

L'intérêt grandissant pour les technologies de l'information en général, et pour l'Internet en particulier, a donné naissance à beaucoup d'architectures réparties, dont certaines reposent sur le paradigme d'agents mobiles. Ce dernier a modifié la façon de faire traditionnelle de déplacer les données vers le code, en déplaçant plutôt le code vers les données sous la forme d'un agent mobile. En effet, un agent mobile peut se déplacer d'un serveur à un autre, interagir localement avec chacun d'eux pour ne récupérer que les données pertinentes avant de revenir chez le client. Cette façon de faire a beaucoup d'avantages par rapport à l'architecture classique dite client/serveur, tels la réduction de la latence et du trafic du réseau et l'exécution asynchrone. De ce fait, beaucoup d'applications utilisant cette technologie ont vu le jour, notamment dans le domaine du commerce électronique. Cependant, la sécurité demeure l'obstacle majeur à son déploiement à grande échelle qui est indispensable à deux niveaux. D'une part, il faut protéger la plate-forme contre des attaques de la part d'agents malicieux et d'autre part, il faut protéger l'agent mobile lui-même des attaques de plates-formes malicieuses. Dans ce dernier cas, il s'agit d'un problème très difficile à résoudre, étant donné que la plate-forme exécutant l'agent mobile a le contrôle total sur celui-ci.

L'objectif principal de ce mémoire est de concevoir un protocole de sécurité permettant de protéger un agent mobile contre les attaques de plates-formes malicieuses, pour une application du commerce électronique. Pour ce faire, nous utilisons le concept d'exécution de référence en exécutant, en parallèle avec l'agent mobile, un clone de ce dernier dans un serveur de confiance. Plus précisément, nous supposons que le réseau est décomposé en plusieurs régions et, dans chaque région, nous supposons l'existence d'un serveur de confiance qui peut exécuter le clone de façon sécuritaire. De cette façon, pendant que l'agent mobile migre d'une plate-forme à une autre pour accomplir sa mission, son clone se déplace d'un serveur de confiance à un autre. Ce dernier ne migre vers un autre serveur de confiance que quand l'agent mobile change de région et migre alors vers le serveur de confiance de la région concernée. L'objectif de notre protocole

est de protéger l'agent mobile contre la modification de son code, de son flux d'exécution, de ses données ainsi que de son itinéraire. Afin de le réaliser, nous avons d'abord recensé les attaques décrites dans la littérature ainsi que les méthodes de protection existantes. Par la suite, nous avons décelé leurs avantages et leurs inconvénients, pour finalement concevoir notre protocole afin de détecter le maximum d'attaques possibles, sans compromettre l'intérêt que procure l'utilisation de la technologie des agents mobiles. À l'issue de cette étape, nous avons décrit les différentes séquences de notre protocole ainsi que les mécanismes qui lui permettaient de détecter les attaques.

Après cette étape de spécification, nous avons choisi la plate-forme Grasshopper pour implémenter un prototype de notre protocole, en utilisant un réseau local de type ethernet. L'application choisie est un agent qui se déplace d'une plate-forme à une autre, à la recherche du meilleur prix d'un certain produit. En plus de l'implémentation d'un agent simple représentant l'application non sécurisée, nous avons implémenté deux scénarios de notre protocole en restant le plus fidèle possible à sa spécification. Le premier scénario est optimiste et suppose que toutes les plates-formes visitées par l'agent mobile appartiennent à la même région du réseau, et nécessite donc l'exécution d'un seul serveur de confiance. Le deuxième scénario est quant à lui pessimiste et suppose que toutes les plates-formes visitées sont dans des régions différentes; il utilise par conséquent un serveur de confiance pour chaque plate-forme visitée.

Finalement, nous avons implémenté notre protocole et nous avons démontré sa capacité à détecter les attaques voulues. De plus, nous avons testé les deux scénarios de notre implémentation et nous avons comparé leurs performances à celles de l'agent mobile simple, i.e. celui de l'application non sécurisée. Les critères de performances qui ont servi de base pour la comparaison sont le trafic généré et le temps total d'exécution. Ces tests ont montré que, pour avoir une application sécuritaire, il fallait payer un certain prix relatif à la performance. Pour notre protocole, ce prix se chiffre à une moyenne de 97% plus de trafic et 87% plus de temps qu'une application non sécurisée. Ces chiffres

restent raisonnables étant donné la solution de sécurité que ce protocole apporte à une application du commerce électronique utilisant un agent mobile.

ABSTRACT

The increasing interest in the information technologies in general and Internet in particular has given rise to a lot of distributed architectures, among which we find the mobile agent paradigm. This one has inverted the traditional way of doing by moving the code up to the data rather than moving data up to the code. Therefore, a mobile agent can move from one server to another, locally interact with each one of them, to finally come back to the client with only the pertinent data. This paradigm has several advantages in comparison with the traditional client/server architecture, like the reduction of network latency and traffic as well as the asynchronous execution. Thus, many applications using this technology have been developed, notably in the electronic commerce area. However, the lack of security framework remains the major obstacle of its large-scale deployment. These security requirements are needed at two levels. In one hand, agent platform has to be protected against malicious mobile agent attacks and on the other, mobile agents has to be protected against malicious platforms' attacks. This latter one is a very hard problem to solve because the platform executing the agent has the complete control on it.

The main objective of this thesis is to design a security protocol that protects a mobile agent against attacks from malicious platforms, for electronic commerce applications. We have used the reference execution concept by executing, in parallel to the mobile agent execution, its clone in a trusted server. More specifically, we assume that the network is composed of several regions within which we assume the existence of a trusted server which can securely execute the clone. Therefore, while the mobile agent migrates from one platform to another to complete its task, its clone migrates from one trusted server to another. This clone migrates to another trusted server only when the mobile agent moves to another region and moves hence to the trusted server of the corresponding region. The purpose of our protocol is to protect the mobile agent against its code, its execution flaw, its data and its itinerary modifications. To achieve this goal, we have first listed the attacks described in the literature and the existing protection

methods. After that, we have identified their advantages and disadvantages to finally design our protocol in order to detect as much attacks as possible, without losing the advantages of using the mobile agent technology. Finally, we have described the different sequences of our protocol along with the mechanisms that allow detecting the attacks.

After this specification step, we have chosen the Grasshopper platform to implement a prototype of our protocol, using an Ethernet local area network. The chosen application is a mobile agent that moves from one platform to another looking for the best price of a certain product. In addition to the implementation of a simple agent representing the insecure application, we have implemented two scenarios of our protocol being the more faithful as possible to its specification. The first scenario is optimistic and assumes that all the visited platforms are located in the same region and therefore requires the execution of only one trusted server. The second scenario is pessimistic and assumes that all the visited platforms are in different regions and consequently involves one trusted server for each visited platform.

Finally, we have implemented our protocol and have proven its capacity to detect the different attacks. Moreover, we have tested the two scenarios of our implementation and have compared their performances to those of the simple mobile agent, i.e., the one used in the insecure application. The performance criteria we have used as basis of the comparison are the generated traffic and the total execution time. These tests have revealed that to make an application secure, we have to pay a price in terms of performance. This price amounts for our protocol to an average of 97% more traffic and 87% more execution time than an insecure application. These results remain reasonable given the security solution that our protocol brings to a mobile agent electronic commerce application.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE I
INTRODUCTION

Le paradigme Agent Mobile apparaît comme une technologie très prometteuse pour développer des applications dans des environnements ouverts, répartis et hétérogènes comme l'Internet. Dans beaucoup de domaines d'applications, comme le commerce électronique et mobile, la recherche d'information, et l'apprentissage électronique et mobile, l'utilisation de la technologie des agents mobiles peut s'avérer très bénéfique. Cette technologie offre plusieurs avantages par rapport à l'architecture classique du client/serveur, comme la réduction de la latence du réseau, l'exécution asynchrone, la robustesse du comportement et la tolérance aux fautes. Cependant, une diffusion plus large de l'utilisation des agents mobiles est actuellement limitée par le manque d'une architecture sécuritaire qui dresse les problèmes engendrés par ces derniers tout en conservant leur efficacité, sujet du présent mémoire. Dans ce chapitre, nous allons tout d'abord définir les concepts de base, puis esquisser les éléments de la problématique afin de prendre conscience des problèmes de la sécurité, nous allons ensuite préciser les objectifs de recherche, et finalement présenter le plan du mémoire.

1.1 Définitions et concepts de base

Il n'y a pas de définition rigoureuse du terme « *agent* », mais dans le cadre de ce mémoire, nous définissons un agent comme un logiciel autonome, qui a un ou plusieurs but, un champ de compétence, qui peut ou non collaborer et communiquer avec d'autres logiciels et utilisateurs. La mobilité n'est pas nécessaire pour dire qu'un logiciel est un agent. Un agent est *mobile* quand il est conçu pour se déplacer d'un ordinateur à un autre. Les *agents mobiles* contiennent non seulement le *code* du programme, mais aussi les *données* et *l'état* d'exécution. Le code peut consister par exemple en des classes Java. Les données sont représentées à travers les valeurs des variables des objets constituant l'agent. L'*état* de l'agent consiste en toute l'information nécessaire à ce

dernier pour qu'il continue son exécution dans la plate-forme suivante (la structure des données, la pile d'exécution, le compteur ordinal, etc.).

Les agents mobiles se déplacent d'un hôte à un autre, sur la base de leur programme logique. En fait, ils ne sont pas mobiles par eux-mêmes, mais ont besoin de la coopération de leur système d'agents mobiles afin de *migrer* d'un hôte à un autre. On peut parler de *mobilité au sens faible* quand seulement le code peut se déplacer d'un hôte à un autre, comme les classes Java dans une applet. La mobilité est plus forte quand le code et les données peuvent se déplacer; elle est supportée par Aglets par exemple. Finalement, à proprement parler, la mobilité au sens fort permet le déplacement du code, des données, ainsi que de l'état d'exécution; elle est supportée par Telescripts, D'Agents et Sumatra, par exemple.

Un agent mobile est basé sur un code qui n'est pas nécessairement écrit par l'utilisateur, mais ce dernier doit se porter garant de ses actes. Le programmeur de l'agent est appelé *créateur* de l'agent, et l'utilisateur de ce dernier, *propriétaire*. L'environnement d'exécution de l'agent est appelé *plate-forme*. L'agent est d'abord lancé de sa plate-forme d'origine (home platform) en qui on suppose qu'il a confiance. Il peut alors migrer vers une autre plate-forme et retourner à sa plate-forme d'origine. Ceci est le cas élémentaire du mono-saut (single hop) qui peut être sécurisé de manière classique, mais n'utilise pas toutes les potentialités des agents mobiles. Dans le cas du multi-saut (multi hop), l'agent migre en chaîne vers plusieurs plates-formes avant de revenir à sa plate-forme d'origine; l'ensemble de ces plates-formes visitées est appelé *itinéraire* de l'agent. Quand un agent veut décider de la plate-forme de sa prochaine migration, soit il choisit une plate-forme de son itinéraire, soit il contacte un répertoire de service (broker) grâce aux informations fournies par la plate-forme courante, ou bien il demande des recommandations à cette dernière.

1.2 Éléments de la problématique

La nature ouverte des environnements d'exécution des agents rend ces derniers vulnérables à des attaques à travers l'Internet. Ainsi, n'importe quel ordinateur, connecté à Internet et possédant une plate-forme d'agents mobiles, peut créer et envoyer des agents mobiles à travers Internet. Les plates-formes hôtes sont ainsi exposées à des attaques de la part d'agents malicieux. Simultanément, l'agent mobile doit être protégé d'une plate-forme malicieuse qui peut altérer sont comportement, espionner des informations confidentielles ou tout simplement ré-exécuter l'agent dans une autre plate-forme. Ainsi, les problèmes de sécurité dans les agents mobiles peuvent être classifiés en deux catégories principales qui sont : la *protection de la plate-forme* et la *protection de l'agent mobile.*

Le problème de l'agent malicieux consiste en général en l'accès non autorisé à la plate-forme. Dans les systèmes informatiques, le contrôle d'accès est implémenté au niveau du système d'exploitation où les utilisateurs sont authentifiés au début de leurs sessions grâce à un mécanisme de mot de passe. Ce schéma est très difficile à étendre aux agents mobiles, puisqu'il est impossible à un agent d'avoir un mot de passe pour chaque plate-forme qu'il visite. L'utilisation de mots de passe non seulement augmente la taille de l'agent, mais exige aussi de s'assurer que ces derniers ne soient pas volés pendant l'exécution ou le transport de l'agent. Quand l'agent arrive sur la plate-forme, il nécessite les privilèges de son utilisateur pour cette dernière pour qu'il puisse exécuter son code. Cette situation engendre de nouvelles formes de menace pour la plate-forme. En effet, le fait que l'agent s'exécute dans la plate-forme en utilisant les ressources de cette dernière, rend ces ressources vulnérables à des attaques. S'il n'y a pas de mécanismes de protection appropriés, l'agent mobile peut collecter ou modifier des informations privées du système. De plus, il peut consommer plus de ressources (mémoire, temps CPU, etc.) et rester plus longtemps que prévu dans la plate-forme. Un tel comportement affecte le temps de réponse de la plate-forme et peut, le cas échéant, mener à un déni de service pour d'autres agents qui s'exécutent sur la même plate-forme. En outre, pendant son exécution, l'agent mobile peut créer un agent stationnaire

qui va résider dans la plate-forme afin de l'espionner. En l'absence de mesures de sécurité appropriées, cet agent peut accéder aux ressources du système et effectuer des attaques de type déni de service.

L'approche généralement adoptée pour résoudre le problème de l'agent malicieux est inspirée de celle déjà existante pour sécuriser les systèmes conventionnels. Elle consiste en l'authentification et la restriction des privilèges des agents mobiles qui visitent la plate-forme. Cependant, des restrictions excessives peuvent nuire à l'agent qui ne pourra pas effectuer sa mission.

Pour ce qui est du problème de la plate-forme malicieuse, le scénario est le suivant. L'agent arrivant sur la plate-forme s'exécute dans l'environnement d'exécution fourni par cette dernière. Les instructions exécutées par cet agent sont converties en des appels système par le système d'exploitation. Ainsi, l'agent mobile, avec son code, son état et ses données, est complètement exposé à la plate-forme qui peut non seulement accéder aux informations contenues dans l'agent, mais aussi modifier ces informations et altérer le comportement futur de celui-ci. De ce fait, la protection des données de l'agent et de son état est un problème très difficile à résoudre. Les attaques auxquelles un agent mobile est exposé sont décrites brièvement dans ce qui suit. Étant donné que l'agent s'exécute dans l'environnement fourni par la plate-forme, cette dernière peut observer toutes les opérations que cet agent effectue, copier les instructions et créer un clone de celui-ci. Ce clone peut être utilisé pour se faire passer pour l'agent original (mascarade). La plate-forme peut aussi se faire passer pour une autre plate-forme assez facilement, en retournant de fausses informations à l'agent (un faux hostname, par exemple). Puisque l'agent n'a aucun moyen de vérifier ces informations, l'identité de la plate-forme peut facilement être déguisée.

Un autre type d'attaque est le déni de service. La plate-forme peut capturer l'agent ou le tuer résultant en la perte de ses données et de son code. Une autre forme de cette attaque est que la plate-forme envoie l'agent à une plate-forme alliée plutôt qu'à une plate-forme concurrente. De plus, quand l'agent s'exécute, la plate-forme peut observer son contenu et modifier ses données, s'il n'y a pas les mécanismes de protection

nécessaires. La plate-forme peut aussi délibérément mal exécuter le code en forçant par exemple une variable à une certaine valeur dans un test conditionnel, sans que l'agent ne s'en aperçoive. Par ailleurs, l'agent peut se faire attaquer pendant son transport, entre deux plates-formes, et sans les mécanismes de sécurité appropriés, l'intrus peut modifier le code et les données de l'agent.

Le problème de la plate-forme hostile est très difficile à résoudre. Parmi les techniques citées dans la littérature on trouve : le calcul de fonctions cryptographiques qui consiste à encrypter le code, la boite noire limitée dans le temps qui permet de cacher le contenu de l'agent pendant un certain temps, ou encore le protocole multi-saut qui utilise une seule clé pour encrypter les données, les autres étant déduites de cette dernière dans chaque plate-forme. Ces techniques ne permettent pas de protéger l'agent de manière absolue mais, mis à part leurs inconvénients, détectent quelques attaques dans des cas particuliers.

1.3 Objectifs de recherche

Ce mémoire à pour objectif principal de concevoir un protocole sécuritaire robuste et tolérant aux fautes permettant de protéger l'agent mobile des attaques malicieuses. Plus précisément, ce travail de recherche vise à :

- concevoir un protocole de protection de l'agent mobile, en s'inspirant des différents mécanismes disponibles dans la littérature ;
- analyser la sécurité de ce protocole en déterminant les types d'attaques qui sont détectés, dans le but de comparer sa capacité de détection avec celle des protocoles existants ;
- implémenter dans une plate-forme d'agents mobiles le protocole proposé, ainsi qu'un agent mobile simple afin de pouvoir les comparer ;
- évaluer l'efficacité de notre protocole et sa capacité de détecter les attaques, en exécutant un plan d'expériences permettant, entre autres, de mesurer son coût et de le comparer avec celui d'un agent simple.

1.4 Plan du mémoire

Ce mémoire comprend cinq chapitres. Le chapitre courant est le chapitre 1 en guise d'introduction. Le chapitre 2 fait le point sur les menaces qui pèsent sur les agents mobiles, ainsi que sur les plates-formes qui les accueillent; il présente aussi les principales méthodes de sécurisation décrites dans la littérature, un modèle général de sécurité, et les efforts de standardisation existant. Le chapitre 3 décrit en détail notre protocole, analyse sa capacité à détecter les différentes attaques, et présente ses avantages et ses inconvénients. Le chapitre 4 présente les détails de l'implémentation de notre protocole et en analyse les résultats. En guise de conclusion, le chapitre 5 présente une synthèse des travaux, les limitations du protocole puis finalement des indications pour des recherches futures.

CHAPITRE II
SÉCURITÉ DES AGENTS MOBILES

La technologie des agents mobiles est une alternative très prometteuse à l'architecture classique du client/serveur. De nombreuses applications basées sur cette technologie ont vu le jour, particulièrement dans le domaine du commerce électronique. Cependant, la sécurité demeure le gros obstacle auquel fait face le déploiement à grande échelle de cette technologie. Dans ce contexte, deux aspects de sécurité sont à considérer, la protection de la plate-forme des attaques d'un agent malicieux et inversement la protection de l'agent mobile contre des plates-formes malicieuses. Plusieurs techniques ont vu le jour afin d'assurer une telle sécurité mais aucune d'elles ne fournit une solution globale permettant l'utilisation à grande échelle de cette technologie. Dans ce chapitre, nous allons tout d'abord présenter les différentes attaques connues dans la littérature auxquelles les agents mobiles et les plates-formes hôtes sont exposés. Par la suite, nous décrirons les techniques existantes pour prévenir ou détecter de telles attaques, leurs avantages ainsi que leurs inconvénients. Suivra un modèle de sécurité montrant concrètement les problèmes de sécurité et les mécanismes de protection, pour finalement présenter les efforts de standardisation qui sont déployés par *FIPA* et *MASIF*,

2.1 Caractérisation des attaques

Dans cette partie nous allons expliquer comment les agents mobiles et les plates-formes hôtes peuvent, intentionnellement ou accidentellement, abuser les uns des autres. Deux grandes catégories d'attaques ont été recensées, l'attaque des plates-formes par les agents mobiles et les attaques des agents mobiles par des plates-formes ou d'autres agents mobiles. Pour chaque type d'attaque, nous allons présenter comment un agent mobile peut abuser de l'information, du logiciel, du matériel ou des ressources de la

plate-forme qui l'accueille et inversement, comment un agent mobile peut être détruit, volé ou altéré par une plate-forme ou par d'autres agents mobiles.

2.1.1 Mascarade

C'est le fait de se faire passer pour un autre ou donner de fausses informations à des fins de tromperies.

Envers un hôte

Un agent mobile peut demander à distance un service à une plate-forme en se faisant passer pour un autre. Ainsi, il peut avoir accès à des ressources auxquelles il n'aurait pas eu accès sous sa vraie identité.

Envers un agent mobile

Une plate-forme peut aussi se faire passer pour une autre en se présentant à l'agent avec une fausse identité. Elle peut, en outre, envoyer l'agent vers une destination autre que celle où il voulait aller, en dehors de son périmètre de sécurité par exemple.

2.1.2 Déni de service

C'est le fait d'entraver, partiellement ou complètement, un ou plusieurs services d'un hôte, ou bien empêcher un agent d'accéder à des ressources ou des services.

Envers un hôte

Un agent mobile peut surcharger de travail une ressource ou un service de la plate-forme, en consommant délibérément trop de CPU, trop de connexions réseau, ou en surchargeant les mémoires tampons d'autres processus.

Envers un agent mobile

La couche d'exécution d'une plate-forme peut empêcher l'agent d'accéder à certains ressources ou services, comme l'accès aux fichiers ou au réseau. Ceci va

entraver la bonne exécution de ce dernier, voire même son déplacement dans le réseau pour accomplir sa mission.

2.1.3 Espionnage

L'espionnage comprend l'accès illégal ou indésiré et le vol d'information d'un hôte ou d'un agent mobile.

Envers un hôte

Un agent mobile peut accéder à une information privée de la plate-forme et l'espionner, en enregistrant secrètement des paquets arrivant vers cette dernière, puis en les transmettant à un site non autorisé. Un agent peut aussi utiliser un canal caché pour envoyer des données de façon masquée, violant ainsi la politique de sécurité de la plate-forme. Ceci peut se faire, par exemple, en alternant son état de disponibilité (occupé/disponible), pour signaler des données en binaire à un collaborateur.

Envers un agent mobile

À n'importe quel moment de sa migration, un agent risque d'avoir certaines informations espionnées par la plate-forme, à moins qu'elles soient encryptées. Cependant, durant son exécution, l'agent pourrait décrypter certaines de ces informations offrant ainsi une fenêtre de vulnérabilité.

2.1.4 Altération

L'altération est la destruction ou la modification des fichiers de la plate-forme hôte, de sa configuration, de son matériel ou d'un agent mobile et sa mission.

Envers un hôte

Un agent mobile peut détruire ou changer les ressources et les services d'un hôte en les reconfigurant, les modifiant, ou en les effaçant de la mémoire ou du disque.

Quand un agent mobile cause des dégâts à une plate-forme, ces derniers se répercutent sur les autres agents qui s'exécutent dans cette plate-forme.

Envers un agent mobile

Un hôte peut détruire un agent mobile en l'effaçant ou en effaçant tout ce qu'il a rassemblé comme information et le laisser dans un état instable. Un agent mobile peut être aussi altéré en modifiant son code, son état d'exécution et ses données, compromettant ainsi sa mission. De plus, des agents mobiles partageant la même couche d'exécution peuvent s'attaquer mutuellement.

2.1.5 Harcèlement

C'est le fait d'ennuyer des usagers avec des attaques répétées ou des messages indésirables.

Envers un hôte

Un agent mobile peut être programmé pour harceler des usagers en affichant sur les écrans des hôtes des photos ou des messages de publicité.

Envers un agent mobile

La plate-forme peut attaquer l'agent de façon à ennuyer son propriétaire en le harcelant pour qu'il dévoile de l'information sur son propriétaire.

2.1.6 Ingénierie sociale

Elle consiste en la manipulation des usagers, des plates-formes ou des agents mobiles en utilisant la désinformation, la coercition ou d'autres moyens.

Envers un hôte

Un agent mobile peut jouer un rôle dans l'ingénierie sociale. Ainsi, il peut par exemple demander le mot de passe d'un utilisateur, en se faisant passer pour l'administrateur du système.

Envers un agent mobile

La plate-forme peut aussi désinformer l'agent dans l'objectif de le manipuler ou manipuler son propriétaire. En effet, elle peut, par exemple, se faire passer pour sa plate-forme d'origine afin qu'elle puisse accéder à des informations confidentielles.

2.1.7 Bombe logique

Elle consiste en la réalisation de n'importe quelle attaque précédemment citée, mais sous l'apparition d'un certain événement extérieur.

Envers un hôte

Un agent mobile peut transporter un code caché qui est programmé pour se déclencher par un événement spécifique (le temps, l'endroit où se trouve l'agent, etc.).

Envers un agent mobile

Un hôte peut attaquer un agent parce qu'il transporte quelque chose en particulier, ou parce qu'il est la propriété d'une certaine personne.

2.1.8 Attaque composée

C'est une attaque complexe qui implique la complicité de plusieurs hôtes ou plusieurs agents mobiles.

Envers un hôte

Des agents mobiles peuvent collaborer afin d'effectuer une ou plusieurs des attaques précédemment citées contre une certaine plate-forme.

Envers un agent mobile

Plusieurs plates-formes ou agents mobiles peuvent collaborer et traquer l'agent afin d'espionner son comportement, ou des informations confidentielles qu'il pourrait transporter.

2.2 Protection de la plate-forme

Sans une protection appropriée, une plate-forme est exposée à des attaques de sources diverses incluant des agents malicieux et des plates-formes malicieuses. La plupart des techniques de protection conventionnelles, utilisées dans les systèmes fiables et dans les communications sécuritaires, peuvent être utilisées pour protéger la plate-forme des attaques malicieuses.

L'une des grandes préoccupations dans l'implémentation d'une plate-forme d'agents mobiles est de s'assurer qu'il n'y ait des interférences ni entre les différents agents s'exécutant sur la même plate-forme, ni entre les agents et la plate-forme elle-même. Ceci devient critique quand la plate-forme, et donc ses mécanismes de sécurité, s'exécutent dans la même couche que l'agent. Dans ce cas, un moniteur de référence est utilisé. Ce dernier est un programme qui contrôle l'accès de l'agent mobile à l'information, aux services et aux ressources de l'ordinateur hôte. Pour ce faire, il se réfère à des instructions contenues dans la politique de sécurité pour déterminer si l'accès doit être autorisé à un agent ou non, selon l'identité et la provenance de ce dernier. La personne responsable du hôte définit cette politique de sécurité pour prévenir les attaques potentielles. Certaines plates-formes utilisent le concept de domaines de sécurité en organisant les ressources et les services en groupes ayant leurs propres politiques de sécurité et listes d'accès. Ceci facilite la gestion de l'accès entre des groupes d'agents mobiles à qui ont veut accorder des niveaux d'accès différents. Différentes politiques de sécurités peuvent être alors appliquées dépendamment de la provenance de l'agent [KAR97].

Parmi les techniques utilisées pour protéger les plates-formes, citées dans la littérature [HAS01], on trouve, l'isolation logicielle, l'interprétation du code, la

signature du code, l'évaluation de l'état, l'historique des hôtes et le code avec preuve. Ces techniques sont présentées plus en détails dans ce qui suit.

2.2.1 Isolation logicielle

Comme son nom l'indique, cette technique [WAH93] est une méthode d'isolation, de façon logicielle, des modules d'application dans des domaines de fautes distincts. Elle permet à des programmes non fiables d'être exécutés de manière sécuritaire dans l'espace virtuel d'adresses d'une application. Ainsi, ces modules non fiables sont transformés de façon à ce que tous les accès à la mémoire soient restreints au code et aux données dans leurs domaines de faute. L'accès aux ressources de la plate-forme peut aussi être contrôlé à travers un identifiant associé à chaque domaine. Cette technique, appelée aussi carré de sable, est très efficace comparée à l'utilisation d'un matériel permettant d'avoir des espaces d'adresses différents pour des modules séparés, quand ces derniers communiquent fréquemment à travers les domaines de faute.

2.2.2 Interprétation du code

L'idée générale de cette méthode est de vérifier le code de l'agent, de façon à savoir s'il est nuisible ou pas, afin d'accepter ou de refuser son exécution. Un des langages d'interprétation les plus utilisés est le langage Java [FUG98] pour sa sécurité renforcée. Il utilise le modèle de sécurité du carré de sable afin d'isoler la mémoire et l'accès aux méthodes et maintenir des domaines d'exécution mutuellement exclusifs. Une vérification statique est effectuée sur le code à exécuter et éventuellement une vérification dynamique pendant son exécution. Un espace de nom distinct est attribué à un code non fiable et l'assemblage des références entre les modules dans deux espaces de noms différents est restreint aux méthodes publiques. Un gestionnaire de sécurité gère tous les accès aux ressources du système, jouant le rôle d'un moniteur de référence. De ce point de vue, Java est considéré comme un langage sécuritaire. Cependant, il a été noté qu'il y avait des limitations concernant le compte rendu pour la mémoire, le CPU et les ressources réseau consommées par des processus individuels, en plus du problème

concernant le support pour la mobilité des processus. En général, les langages de scripts couramment utilisés n'incorporent pas un modèle de sécurité explicite dans leur design, la décision est plutôt prise pendant l'implémentation.

2.2.3 Signature du code

Une technique fondamentale pour protéger une plate-forme est la signature numérique du code ou d'autres objets. Une signature numérique sert à confirmer l'authenticité d'un objet par rapport à son origine ainsi que son intégrité. Typiquement, le signataire du code est soit le créateur de l'agent, son utilisateur ou une entité ayant manipulé l'agent [TAR96] [GRA96] [KAR97] [KAR98]. La signature du code implique l'utilisation de la cryptographie à clé publique, qui est basée sur une paire de clé associée à une entité. Une clé est gardée privée et l'autre est rendue disponible publiquement. La signature numérique utilise donc l'infrastructure à clé publique et les certificats numériques. Ces derniers servent à confirmer la correspondance entre l'identité d'une entité et sa clé publique. Cette signature est effectuée en appliquant au code, en l'occurrence, une valeur de hachage à sens unique (i.e. avec une fonction non inversible), puis en encryptant le résultat avec la clé privée du signataire. Ainsi, cette signature sert non seulement à authentifier l'origine du code mais assure aussi son intégrité. En effet, la plate-forme recevant le code authentifie le signataire en utilisant la clé publique de ce dernier. L'intégrité du code est vérifiée en lui appliquant la même fonction de hachage qui lui a été appliquée par son propriétaire, et en vérifiant l'égalité du résultat avec la valeur fournie.

2.2.4 Évaluation de l'état

Le but de cette méthode [FAR96] est de s'assurer qu'un agent n'a pas été altéré, d'une manière ou d'une autre, dû à une modification de son état. Le succès de la méthode repose sur la pertinence de la fonction d'évaluation de l'état à prédire si l'état de l'agent a été modifié ou non. Ces fonctions d'estimation vont servir à la plate-forme

pour déterminer quels privilèges attribuer à l'agent, selon des critères conditionnels relatifs à son état.

Ainsi, un agent peut se voir attribué aucun privilège, ou des privilèges restreints, selon le critère que son état a violé. Le créateur et l'utilisateur de l'agent créent leurs propres fonctions d'estimation et les joignent à l'agent. Étant donné que ce dernier est signé numériquement par chacun d'eux, leurs fonctions sont protégées contre les modifications éventuelles. La plate-forme recevant l'agent vérifie que son état est correct et détermine en conséquence, et selon sa politique de sécurité, les privilèges à attribuer à ce dernier. Cependant, il n'est pas clair comment cette méthode va fonctionner en pratique étant donné que la plage de valeurs prises par l'état de l'agent peut être très grande. Ainsi, cette technique peut bien fonctionner pour des attaques évidentes, mais peut s'avérer inefficace pour détecter des attaques plus subtiles. De plus, les développeurs de cette technique avouent qu'il n'est pas toujours possible de distinguer entre des résultats normaux et des résultats erronés.

2.2.5 Historique de hôtes

L'idée générale de cette technique [ORD96] [CHE95] est de maintenir une liste authentifiée des plates-formes visitées par l'agent tout au long de son parcours. De cette façon, la plate-forme visitée peut décider si elle l'exécute et quels privilèges lui attribuer. Afin d'établir cette liste, chaque plate-forme que l'agent visite rajoute son identité ainsi que celle de la prochaine plate-forme, le tout est signé numériquement conjointement avec l'historique cumulé par l'agent jusqu'à présent. Ainsi, quand la plate-forme suivante reçoit l'agent, elle peut lui attribuer un niveau de confiance selon l'identité des plates-formes visitées par ce dernier. Cette technique n'empêche pas une plate-forme d'être malicieuse, mais permet de la retracer puisqu'elle a effectué une signature numérique de l'agent et ne peut pas le répudier. Un inconvénient majeur de cette méthode est que la taille de l'historique augmente avec le nombre de hôtes visités et par conséquent, la vérification de l'historique dans chaque plate-forme devient fastidieuse et

coûteuse. De plus, la taille de l'agent grandit, ce qui compromet son avantage et son efficacité.

2.2.6 Code avec preuve

L'approche suivie par cette technique [NEC96] oblige l'auteur du code à prouver formellement que ce dernier possède des propriétés de sécurité, précédemment stipulées par l'utilisateur du code, à travers la politique de sécurité de sa plate-forme, par exemple. Cette technique est une technique préventive contrairement à la signature du code qui est une technique d'authenticité et d'identification, et qui ne prévient pas par conséquent l'exécution d'un code malicieux. Le code et la preuve sont envoyés conjointement à l'utilisateur qui peut vérifier les propriétés de sécurité. Un prédicat de sécurité représentant la sémantique du programme est généré directement à partir du code original, afin d'assurer la correspondance avec ce dernier. La preuve est structurée de manière à ce que la vérification soit facile à effectuer, sans utiliser des techniques de cryptographie ou une assistance extérieure. Une fois vérifié, le code peut être exécuté sans vérification supplémentaire. Toute tentative d'altération de ce dernier ou de la preuve de sécurité aboutit soit à une erreur de vérification, soit à une transformation du code sécuritaire.

Des travaux de recherche ont démontré l'applicabilité de cette technique pour le contrôle d'accès à la mémoire, dans le cas de fine granularité ainsi que pour d'autres types de politiques de sécurité, comme le contrôle d'accès aux ressources. Ainsi, cette technique peut être utilisée dans quelques applications comme alternative raisonnable à la technique d'isolation logicielle du code. Les notions utilisées dans cette théorie sont basées sur des principes bien établis de logique, de théorie des types et de vérification formelle. Cependant, il y a potentiellement quelques problèmes à résoudre avant que cette approche ne soit considérée en pratique. Ces problèmes incluent un formalisme standard pour établir la politique de sécurité, une assistance automatisée pour la génération des preuves, et des techniques pour limiter la taille potentiellement grande

des preuves qui peuvent se présenter en théorie. De plus, l'applicabilité de cette technique peut être limitée par les capacités d'exécution de l'utilisateur du code.

2.3 Protection de l'agent

Protéger un agent mobile de son environnement d'exécution est connu pour être un problème très difficile à résoudre. Ceci est dû au fait que chaque ligne du code de l'agent doit être lue, interprétée et exécutée par la plate-forme. De plus, toute l'information contenue dans l'agent, i.e. son code, son état et ses données est visible à la plate-forme, ce qui est nécessaire pour qu'elle puisse l'exécuter. Chaque plate-forme accueillant l'agent a le contrôle total sur son code et ses données et peut les altérer pour son propre avantage. Par exemple, si un agent est utilisé dans le commerce électronique pour acheter un billet d'avion, une plate-forme malicieuse peut modifier ses données de façon à ce que l'achat se fasse chez-elle. Les principaux points de sécurité qui doivent être garantis pour un agent sont l'intégrité et la confidentialité des données, l'intégrité et la confidentialité du flot d'exécution, la non répudiation des données de l'agent provenant d'une plate-forme spécifique, la non répudiation de son exécution et finalement la disponibilité de l'agent.

2.3.1 Traces cryptographiques

La technique des traces cryptographiques [VIG97] permet de détecter l'altération par une plate-forme du code de l'agent, de son état et de son flot d'exécution. C'est un mécanisme de non répudiation par le fait qu'une plate-forme ne pourra pas nier avoir exécuté un certain agent, s'il a résulté en une certaine trace. L'ensemble des traces recueillies par un agent, durant ses exécutions en dehors de sa plate-forme d'origine, constitue l'historique de son exécution. La technique se présente comme suit.

Quand une plate-forme A reçoit un agent de la plate-forme d'origine, et veut l'envoyer vers une plate-forme B, après l'avoir exécuté, elle envoie un message signé contenant son code p, son état S_A, et sa trace d'exécution T_A. B sauvegarde ce message, ainsi, A ne peut pas par la suite nier avoir envoyé l'agent. B envoie ensuite un reçu signé

consistant en un hachage du message reçu, accusant réception de p, S_A et T_A. Ce reçu est stocké par A. La plate-forme B exécute ensuite l'agent, ce qui résulte en un nouvel état S_B et une nouvelle trace T_B. B signe ensuite le tout (p, S_B et T_B), puis l'envoie à la prochaine plate-forme, C. De la même façon que B, C sauvegarde ce message et envoie un accusé de réception signé à B, que ce dernier sauvegarde. A possède donc la preuve que B a bien reçu de sa part p, S_A et T_A, et B la preuve que C a reçu de sa part p, S_B et T_B. Ce processus continue jusqu'à ce que l'agent revienne à sa plate-forme d'origine.

Si le propriétaire de l'agent a des soupçons quant à l'exécution de son agent et pense qu'il a été altéré, il demande les traces cryptographiques à toutes les plates-formes du parcours de l'agent. La vérification commence à la première plate-forme A, qui a la preuve d'avoir envoyé p, S_A et T_A à B. Le propriétaire exécute donc l'agent et vérifie s'il résulte effectivement en l'état S_A et la trace T_A, en s'exécutant normalement. Si ce n'est pas le cas, cela veut dire que l'agent a été altéré par A. Sinon, B doit donner au propriétaire les preuves d'exécution de l'agent comme décrit précédemment. Le processus de vérification continue de cette façon pour toutes les plates-formes du parcours de l'agent, jusqu'à l'état atteint à la plate-forme d'origine. Le gros problème de cette méthode est qu'elle est coûteuse en temps d'exécution, à cause du fait que beaucoup de traces doivent être sauvegardées et beaucoup de messages échangés. De plus, il faut attendre la fin de l'exécution de l'agent pour détecter une éventuelle attaque.

Cette méthode a été adaptée par Hohl [HOH00] pour pallier certains de ces problèmes. Ainsi, il propose qu'après son exécution sur une plate-forme, l'agent soit envoyé avec les variables d'état et les traces précédant et suivant l'exécution. De cette manière, la vérification de la bonne exécution sur la plate-forme précédente peut se faire directement dans la plate-forme suivante. Cependant, ceci ne sera pas efficace dans le cas de deux plates-formes hostiles qui collaborent et qui sont consécutives dans l'itinéraire de l'agent.

2.3.2 Encapsulation des résultats partiels

Quand l'agent migre d'une plate-forme à une autre, il transporte avec lui les résultats de son exécution dans les plates-formes précédentes. Ces résultats partiels peuvent être, par exemple, l'offre la moins chère, dans le cas d'un agent cherchant le meilleur prix d'un certain produit. Si ces résultats ne sont pas protégés, une plate-forme hostile peut effacer ou modifier toutes les offres qui sont meilleures que la sienne. L'encapsulation des résultats partiels permet d'aider à détecter la modification ou l'élimination de ces résultats. Si l'itinéraire de l'agent est connu d'avance, il est suffisant que chaque plate-forme de cet itinéraire signe numériquement son résultat partiel et/ou l'encrypte avec la clé publique du propriétaire de l'agent. En général, cet itinéraire n'est pas connu d'avance. De plus, quelques hôtes peuvent être inaccessibles et engendrer des manques dans la liste des résultats partiels, provoquant ainsi des soupçons.

Dans un mécanisme basé sur le code d'authentification de message, *MAC* (Message Authentification Code), proposé par Yee dans [YEE99], une clé k_1 est donnée à l'agent mobile avant qu'il ne quitte sa plate-forme d'origine. Ce dernier utilise cette clé pour calculer le code d'authentification des résultats partiels, $PRAC_1$ (Partiel Result Authentification Code), de l'offre de la première plate-forme dans laquelle il s'est exécuté. Une fois ce code calculé, il génère une autre clé $k_2=f(k_1)$ et détruit $k1$, f étant une fonction non inversible. À noter qu'il est aussi dans l'intérêt de la plate-forme courante que k_1 soit détruite, car elle n'a pas envie qu'une autre plate-forme modifie son offre. La clé k_2 est utilisée pour calculer le $PRAC_2$ de l'offre de la deuxième plate-forme, et ainsi de suite. Toutes les valeurs du *PRAC* sont rajoutées aux données de l'agent. Quand ce dernier revient finalement à sa plate-forme d'origine, son propriétaire peut vérifier la validité des *PRAC* puisqu'il connaît k_1 et il peut ainsi détecter si une offre a été modifiée ou effacée. Il est vrai qu'avec cette méthode les résultats partiels calculés dans des plates-formes de *1* à *i* ne peuvent pas être modifiés pas une plate-forme malicieuse *j* $(i<j)$. Le problème avec le *PRAC* cependant, est que la plate-forme hostile *j* peut lire la valeur de la clé k_j contenue dans l'agent, et calculer ainsi les valeurs subséquentes des clés k_l $(l>j)$. Si l'agent revient plus tard dans cette plate-forme, elle

peut modifier son offre ainsi que toutes les offres des plates-formes suivantes. Ceci est dû au fait que les résultats partiels ne sont pas protégés.

La technique du *PRAC* a été améliorée dans [KAR98b] pour assurer l'intégrité des résultats partiels. Dans le *PRAC* amélioré, aucune plate-forme n'est capable de modifier les résultats d'aucune autre plate-forme dans le cas d'une visite ultérieure, même pas les siens. Ceci est effectué en réalisant une chaîne avec les valeurs de hachage des résultats partiels, de façon à ce que le résultat partiel obtenu à chaque plate-forme est relié à l'identité de la plate-forme suivante et est signé numériquement. La chaîne de hachage est obtenue en appliquant une fonction de hachage cryptographique à sens unique, comme *SHA-1* par exemple. Plus précisément, la plate-forme d'origine P_0 choisit un nombre aléatoire r_0 et calcule O_0, qui est envoyé à la plate-forme P_1. $O_0 = D_{p0}(E_{p0}(o_0,r_0),h(r_0,P_1))$, o_0 représente l'instance de l'agent dans la plate-forme d'origine, $D()$ représente une opération de signature, $E()$ une opération d'encrytption à clé publique et $h()$ une opération de hachage. Ensuite, chaque plate-forme suivante P_i, choisit un nombre aléatoire r_i et protège ses résultats partiels o_i en les encapsulant de la façon suivante : $O_i = D_{p0}(E_{p0}(o_i,r_i),h(O_{i-1},P_{i+1}))$. Le fait d'inclure $h(O_{i-1},P_{i+1})$, permet au résultat partiel d'être chaîné au résultat partiel précédent, et à l'identité de la plate-forme suivante. Une plate-forme P_i envoie tous les résultats partiels précédents O_k, $(k=0,\ldots,i)$ à la plate-forme suivante P_{i+1}.

Cette méthode ainsi modifiée assure l'intégrité, la confidentialité et la non répudiation des résultats partiels. De plus, le propriétaire de l'agent peut vérifier, une fois son agent de retour, sa bonne exécution et désigner la plate-forme fautive en cas d'attaque grâce aux O_k. Cependant, la taille de l'agent augmente très vite à chaque migration et peut compromettre l'intérêt de son utilisation, si le nombre de plates-formes visitées est trop grand.

2.3.3 Environnement de génération de clés

Cette méthode, telle que proposée dans [RIO98], permet à l'agent de prendre des actions prédéfinies, quand certaines conditions sont vérifiées dans sa plate-forme d'exécution. Cette méthode est alors centrée sur la construction d'agents de façon à ce que, quand une certaine condition est vérifiée, une clé est générée. Cette dernière sert alors à décrypter un code qui sera par la suite exécuté. La condition qui déclenche le mécanisme est cachée soit par une fonction de hachage, soit à travers un mécanise d'encryptage. Cette technique assure ainsi que, ni la plate-forme, ni un observateur extérieur ne peut découvrir le message déclencheur simplement en lisant le code

Supposons que l'agent transporte une chaîne de caractère N et une valeur de hachage $H=h(N \oplus S)$. Il cherche alors la plate-forme qui contient la chaîne de caractère s qui vérifie $h(N \oplus s)= H$. Le propriétaire de l'agent ne veut pas dévoiler ce que l'agent cherche, car peut révéler ses intentions à un concurrent, par exemple. L'agent calcule alors $h(N \oplus s)$ pour chaque chaîne de caractère s et compare le résultat à H. Quand la bonne valeur est trouvée, l'agent calcule la clé de décryptage $h(s)=h(S)$, décrypte et exécute un code, puis envoie par exemple un mail au propriétaire lui indiquant que s vérifiant H a été trouvée dans la plate-forme courante P.

Une faiblesse de cette méthode est qu'une plate-forme qui contrôle complètement l'agent peut tout simplement le modifier pour qu'il affiche son code quand la condition est réalisée au lieu de l'exécuter. Une autre faiblesse est que la plate-forme va devoir limiter les ressources à ce code puisqu'il est généré dynamiquement, et son exécution sera considérée donc comme étant non sécuritaire. Cette technique peut cependant être utilisée conjointement avec d'autres mécanismes de protection, pour des applications spécifiques, et dans des plates-formes appropriées.

2.3.4 Calcul avec des fonctions cryptographiques

Le but du calcul avec des fonctions cryptographiques [SAN98] est de fournir aux agents mobiles une confidentialité de calcul pour les primitives cryptographiques. Supposons que l'agent veuille signer numériquement des données fournies par la plate-

forme qui l'accueille. Même si la clé privée de l'agent est encryptée, il faut qu'il l'a décrypte avant d'effectuer la signature, ce qui permettrait alors à la plate-forme de l'obtenir. L'approche décrite par cette méthode pour éviter ce problème est d'encrypter la fonction qui permet d'effectuer la signature. Le résultat de l'encryptage de la fonction f est la fonction $E(f)()$, qui permet d'avoir un résultat encrypté. Le programme qui permet d'effectuer ces calculs se trouve dans l'agent. Ainsi, pour un résultat x produit dans la plate-forme, le résultat signé est $E(f)(x)$. Ce résultat ne peut être décrypté que par le propriétaire de l'agent, puisqu'il est le seul à connaître $E^{-1}(f)$. Pour le moment, la technique fonctionne seulement avec des fonctions polynomiales ou rationnelles, ce qui rend son utilisation limitée.

2.3.5 Boite noire limitée dans le temps

Avec cette technique, le code d'un agent mobile est transformé en une « *boite noire* » qui a la même fonctionnalité que l'agent, mais qui est plus difficile à analyser [HOH98]. Il n'y a pas un algorithme prédéfini pour utiliser cette technique. Le calcul avec des fonctions cryptographiques pourrait par exemple être utilisé. Puisque le code ne peut pas être protégé de manière absolue de cette façon, on suppose alors l'existence d'un intervalle où l'agent est protégé, i.e. jusqu'à ce que la boite noire ne soit plus fonctionnelle. Cette technique peut être utilisée pour protéger le code et les données dans une certaine période de validité, avec une date d'expiration par exemple, dans le cas de monnaie électronique. Elle ne peut cependant pas être utilisée pour des données qui ont une longue durée de vie, comme les numéros de cartes de crédit. Malheureusement, cette technique est difficile à appliquer en pratique, car il n'y a pas d'approche générale garantie par un algorithme pour quantifier l'intervalle de protection.

2.3.6 Matériel résistant aux altérations

En général, le propriétaire de l'agent mobile ne fait pas confiance aux plates-formes que son agent va visiter. L'idée de cette méthode est d'introduire un composant matériel résistant aux altérations (tamper-resistant hardware) dans chaque plate-forme,

qui assure que l'agent s'exécute sans qu'il soit accédé ou modifié illégalement par cette dernière [YEE99]. Ce composant constituerait alors l'environnement d'exécution de l'agent sur lequel la plate-forme n'a aucun contrôle, et qui interagirait avec lui en client/serveur. La résistance aux altérations est telle que, la plate-forme devrait perdre plus de temps/argent à essayer de rompre le mécanisme de protection qu'elle n'en gagnerait en le rompant effectivement. Un design détaillé d'un système d'agents mobiles avec un composant matériel résistant aux altérations, permettant un échange d'agents mobiles sécuritaire, et basé sur un protocole à clé publique est décrit dans [WIL99].

Cependant, une solution basée sur un tel composant matériel est très coûteuse et n'est pas viable à grande échelle, étant donné que ça nécessiterait que toutes les plates-formes en soient équipées. Une approche moins coûteuse serait d'exécuter seulement une partie de l'agent mobile dans un petit composant matériel résistant aux altérations. Une telle approche, utilisant des cartes à puces comme environnement d'exécution, a été proposée dans [FUN99]. Malheureusement, elle est limitée par le fait que la plate-forme peut contrôler la communication entre la partie de l'agent s'exécutant dans son environnement d'exécution, potentiellement hostile, et celle, sécuritaire, s'exécutant dans la carte à puce. Ainsi, lors des interactions avec la partie du code s'exécutant dans la carte à puce, la plate-forme peut rajouter des informations en lui faisant croire que c'est le résultat effectivement trouvé par l'agent. La plate-forme peut rajouter par exemple des articles plus chers et les faire signer par la partie s'exécutant dans la carte à puce. Ce problème peut être partiellement résolu si cette dernière rajoute des limitations pour chaque commande avant de la signer (par exemple, valide pour seulement des achats inférieurs à 100 $) [WIL99].

2.3.7 Agents coopérants

Les techniques utilisant des agents coopérants ont leurs racines dans le domaine de la tolérance aux fautes. Dans le cas où un seul agent est lancé pour accomplir une certaine tâche, il a en général beaucoup de chance de se faire attaquer par une plate-forme hostile. L'approche décrite dans [ROT99] suppose qu'il y a deux groupes

indépendants de plates-formes qui ne collaborent pas ensemble. Pour accomplir une certaine tâche, deux agents coopérants sont lancés dans le réseau. Le premier agent migre seulement dans le premier groupe de plates-formes, et le second, seulement dans les plates-formes du deuxième groupe. Ceci est montré à la Figure 2.1 extraite de [ALL01]. Un des deux groupes peut être choisi de sorte à ce qu'il ne contienne que des hôtes fiables, de cette façon, un des deux agents s'exécute seulement dans des plates-formes fiables. Ces techniques ne fonctionnent que si on suppose que deux plates-formes de deux groupes différents ne coopèrent pas. Si cette supposition n'est pas réaliste, il y a d'autres schémas possibles mais plus complexes, qui impliquent une tierce partie fiable.

Dans un scénario de deux agents coopérants, un agent protège l'autre des manipulations de son itinéraire à travers le réseau. Ceci est spécialement important si cet itinéraire n'est pas prédéfini par leur propriétaire, auquel cas, il est alors très difficile de détecter des tentatives de manipulation. En fait, pour chaque migration, un agent envoie à l'autre un message contenant l'identité de la plate-forme précédente, de la plate-forme courante, ainsi que de la plate-forme suivante. Dans un autre scénario, chaque agent contient une partie de la monnaie électronique et les deux parties sont nécessaires pour effectuer un paiement valide. Par exemple, le premier agent peut obtenir une offre de la part d'une plate-forme et l'envoie au deuxième via un canal sécuritaire. Ce dernier peut vérifier si l'offre satisfait tous les critères. Si c'est le cas, il envoie sa partie de la monnaie électronique de sorte à ce que le premier agent puisse payer pour les articles indiqués dans l'offre.

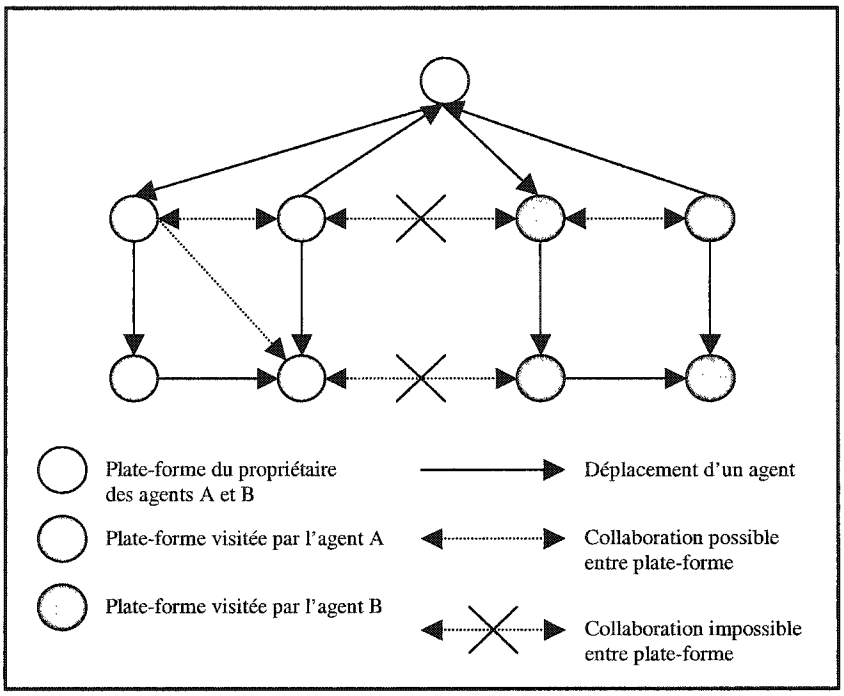


Figure 2.1 Hypothèses de Roth sur les collaborations entre plates-formes

Dans son protocole d'enregistrement d'itinéraire, Roth n'a pas précisé comment l'itinéraire de l'agent coopérant était défini. De plus, il a évoqué le problème d'interversion possible de deux agents coopérants qui effectueraient leurs vérifications dans la même plate-forme. Allée [ALL01] a proposé une amélioration sur ces deux points en déplaçant l'agent coopérant *b* à chaque déplacement de l'agent mobile *a*. En fait, avant de se déplacer, l'agent *a* envoie les informations concernant l'identité de la plate-forme courante et suivante. Quand l'agent *b* les reçoit, il vérifie la validité de l'itinéraire, l'enregistre et se déplace à son tour, et attend que l'agent *a* communique avec lui à nouveau. Pour ce faire, il suppose que l'agent mobile *a* possède une liste ordonnée des plates-formes que son agent coopérant *b* doit visiter.

2.3.8 Copies d'agents

Si la tâche à effectuer par l'agent est idempotente, i.e. n'est pas affectée par le nombre de fois qu'elle est effectuée, plusieurs copies de l'agent peuvent être lancées dans le réseau [SCH97]. Par exemple, la recherche du meilleur prix pour un article est idempotente, mais le paiement du produit ne l'est pas. De cette façon, même si quelques copies sont corrompues, les autres vont effectuer la tâche correctement. Pour vérifier quelles copies sont corrompues et lesquelles ne le sont pas, des schémas de vote sont utilisés. Cette technique de tolérance aux fautes fonctionne avec la supposition que la majorité des plates-formes de migration ne sont pas hostiles.

Une des méthodes proposées dans [SCH97] est basée sur le schéma de partage de secret. Dans un tel schéma *(n, k)*, un secret est partagé en n morceaux de façon à ce que au moins k d'entre eux sont nécessaires pour reconstituer le secret, mais *(k-1)* ou moins ne révèlent aucune information sur le secret. On suppose qu'il y a l plates-formes sur l'itinéraire, incluant la plate-forme d'origine qui est au début et à la fin du parcours. Le protocole fonctionne selon les étapes suivantes, comme illustré à la Figure 2.2.

- *Étape 1 :* La plate-forme d'origine connaît le secret s et le divise en utilisant un schéma de partage *(2k-1, k)*. *2k-1* morceaux, $p_1,\ldots, p_{2k-1}$, sont envoyés à *2k-1* plates-formes, et une copie de l'agent est envoyée avec chaque morceau.
- *Étape 2 :* Chaque plate-forme divise le morceau qu'elle a reçu en appliquant à son tour un schéma de partage *(2k-1, k)*. La plate-forme i envoie chacun des *2k-1* morceaux, $p_{2,i,1},\ldots, p_{2,i,2k-1}$, avec une copie de l'agent à une plate-forme de l'étape 3. Le triplet dans les indices indique, le numéro de l'étape, la plate-forme générant les morceaux, et le numéro du morceau.
- *Étape 3 :* Jusqu'à l'étape *l-2*, chaque plate-forme à l'étape j concatène tous les morceaux qu'elle reçoit des plates-formes du niveau *j-1*. La concaténation effectuée par une plate-forme i est un nouveau secret $(p_{j-1,1,i} \,||\ldots||\, p_{j-1,2k-1,i})$, qui est lui-même divisé en *2k-1* morceaux, $p_{j,i,1},\ldots, p_{j,i,2k-1}$, en appliquant un schéma de partage *(2k-1,k)*. Chaque morceau est envoyé à l'étape suivante avec une copie de l'agent, à chacune des *2k-1* plates-formes de l'étape suivante.

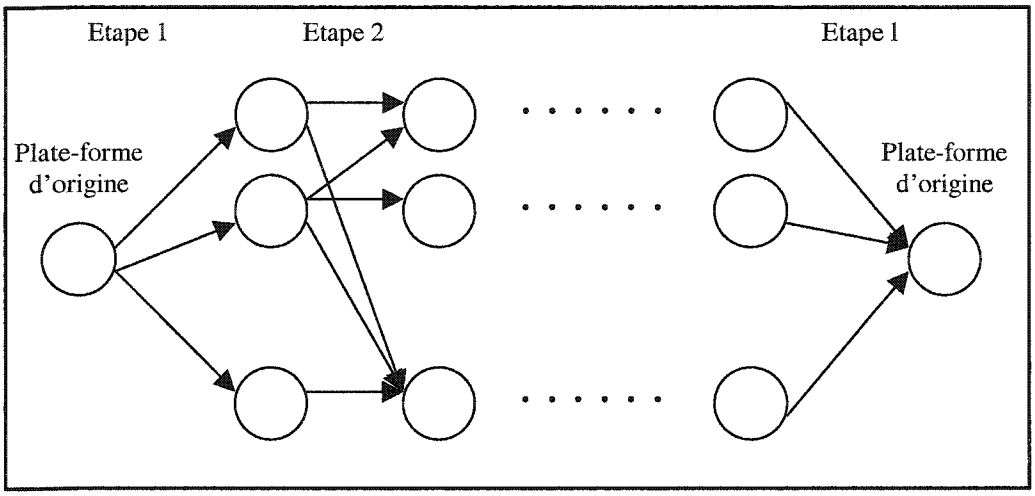


Figure 2.2 Représentation de la technique de copie d'agents

- *Étape l-1 :* Chaque plate-forme à l'étape *l-1* concatène tous les morceaux reçus des plates-formes de l'étape *l-2*. La concaténation composée par la plate-forme *i* est un nouveau secret $(p_{l-2,1,i} || \ldots || p_{l-2,2k-1,i})$, qui est envoyé à la plate-forme d'origine.
- *Étape l :* La plate-forme d'origine reçoit les *2k-1* concaténations $(p_{l-2,1,i} || \ldots || p_{l-2,2k-1,i})$, $i=1,\ldots,2k-1$, de toutes les plates-formes de l'étape *l-2*. Elle applique d'abord le schéma de partage pour récupérer la concaténation de l'étape *l-2*, en enlevant les morceaux de la première position dans chaque concaténation. Ensuite, elle continue à appliquer le schéma inverse pour récupérer le secret d'origine *s*.

Ce protocole fonctionne même si *k-1* plates-formes hostiles ou moins collaborent à chaque étape. Si le secret d'origine ne peut pas être retrouvé, cela veut dire que la majorité des plates-formes ont collaboré, ce qui n'a pas beaucoup de chance d'arriver en général. Pour prévenir la destruction des morceaux du message par une plate-forme hostile, un schéma de partage de secret vérifiable devrait être utilisé. Un autre problème est que le nombre de morceaux augmente à chaque étape de calcul. Ceci peut être évité en utilisant un partage de secret proactif, comme suggéré dans [SCH97].

2.4 Modèle de sécurité d'un système d'agents mobiles

Dans cette partie, nous allons présenter un exemple de modèle de sécurité d'une plate-forme d'agents mobiles [VAR00]. Dans le cas des agents mobiles, comme souvent dans un environnement réparti, le créateur et/ou l'utilisateur délègue à l'agent une certaine action qu'il doit effectuer pour lui. Dans ce contexte, au moins deux questions doivent être posées :

- Est-ce que c'est bien le créateur et/ou l'utilisateur qui a effectivement transféré les privilèges à l'agent ?
- Est-ce que c'est effectivement ce même agent qui effectue la requête ?

Ces deux questions correspondent à l'autorisation et à l'authentification de l'agent. De plus, il peut être important dans plusieurs cas de figure que la plate-forme et l'agent puissent prouver qu'il y a eu telle transaction, tel jour à telle heure, pour la non répudiation. Le but de cette section est de présenter un modèle général de sécurité pour le design et l'implémentation d'une plate-forme sécuritaire. Le modèle proposé [VAR00] permet l'authentification des agents par la plate-forme, l'autorisation des agents basée sur leurs privilèges, l'intégrité et la confidentialité des agents dans les communications. Ce même modèle a été étendu pour traiter les problèmes de délégation et de clonage, mais cette partie n'est pas présentée dans cette section. Le modèle se présente comme suit.

Chaque plate-forme a un composant de confiance de gestion de la sécurité appelé *SMC* (Security Management Component). Des plates-formes qui obéissent à la même politique de sécurité sont groupées ensemble dans un même domaine. Chaque domaine a une autorité de gestion de la sécurité appelée *SMA* (Security Management Authority). Le *SMA* interagit avec les *SMCs* de son domaine pour l'établissement et la maintenance de la politique de sécurité, et interagit avec des *SMAs* d'autres domaines dans la situation d'inter-domaines. Quand l'agent arrive à la plate-forme pour s'exécuter, le *SMC* joue le rôle de moniteur de référence et détermine si une certaine action, demandée par l'agent, est permise ou non. La politique de sécurité du *SMC* spécifie les conditions sous

lesquelles une requête est acceptée. Elle peut être basée sur l'identité du créateur de l'agent, de son envoyeur, de la fonction du code du programme, de son état et de l'application visée.

Du côté de l'agent, on considère qu'il transporte avec lui un passeport de sécurité qui encapsule non seulement les actions à effectuer, mais aussi les privilèges et d'autres attributs de sécurité pour effectuer ces actions. L'agent contient ainsi toutes les informations nécessaires pour que la plate-forme puisse prendre la décision d'autorisation. On appel un tel agent un *AeS* (Agent Enhanced Security). La structure d'un tel agent est comme suit :

- *Identifiant : (Id_SeA, certificat_créateur, certificat_SMC_créateur, timestamp, durée_vie)*
- *Privilèges : {<N_Id, privilège, timestamp, durée_vie>}*
- *Code_agent : (Code_sécurité, code_application)*
- *Données : (Données, itinéraire)*
- *Tags_sécurité : (Tag_securité-C, tag_sécurité-S)*

Voici une description de chaque attribut :

Identifiant

Chaque agent a un *Id_SeA* unique, un nombre par exemple, qui lui est attribué au moment de sa création. Le certificat du créateur se réfère à l'identité du créateur de l'agent. Il contient ainsi son identité, sa clé publique et la durée de validité du certificat; le tout est signé par le *SMC* en utilisant sa clé privée. Le certificat *SMC* du créateur identifie le hôte et il est signé par le *SMA*. Le « *timestamp* » identifie la date de création de l'agent, et *durée_vie*, sa durée de vie.

Privilèges

Cette section spécifie les privilèges de l'agent qui lui ont été attribués par son créateur. Elles sont décrites en terme d'opérations ou de méthodes applicables à certains

objets d'une certaine classe. Ces privilèges vont permettre à l'agent d'exécuter sa tâche, mais seulement si la politique de sécurité de la plate-forme lui permet. Le *Id_N* est le numéro associé au privilège, le « *timestamp* » indique la date de création du privilège, et *durée_vie* indique sa durée de vie.

Code_agent

Il consiste en des méthodes exécutables. Il y a deux types de codes, un de l'application elle-même, qui définit la tâche de l'agent et l'autre, de sécurité, qui consiste en des méthodes qui sont rajoutées automatiquement lors de la création de l'agent. Ce code est responsable de la génération et la maintenance des paramètres de sécurité. Il contient deux méthodes, une méthode *itinéraire* qui crée une liste des identités des plates-formes visitées et une méthode *checksum* qui sert à calculer les hachages cryptographiques nécessaires pour la génération des étiquettes de sécurité (c.f. *Tag-sécurité-C* et *Tag-sécurité-S*).

Données

Le champs des données contient l'état de l'agent, en terme de valeurs des variables et des variables initiales. Il contient aussi les résultats générés après son exécution. Ces résultats peuvent être protégés pour la confidentialité, l'intégrité et l'authentification de leurs sources, de sorte à ce que seulement l'envoyeur puisse lire ces données et vérifier leurs sources, une fois l'agent revenu dans sa plate-forme d'origine. Quant au champ *itinéraire*, il consiste en une liste des plates-formes visitées par le *SeA*. Il est créé par la méthode *itinéraire* décrite précédemment.

Tag_sécurité-C

Il est généré par le créateur de l'agent. Il contient le résultat de hachage *hash-c* du *SeA* original, signé par son créateur avec sa clé privée. La valeur *hash-c* est calculée sur l'identifiant, les privilèges, le code de l'agent (de l'application et de sécurité), et les valeurs fixes de départ contenues dans le champs des données. Cette étiquette de sécurité

permet aux différentes plates-formes visitées de vérifier l'intégrité du code et des données de départ, en plus d'authentifier le créateur.

Tag_sécurité-S

Il est généré par le client qui envoie l'agent. Il contient la valeur de hachage *hash-s,* signée par l'envoyeur avec sa clé privée. Cette valeur *hash-s* est calculée sur *hash-c*, les données et le contenu de la requête, qui elle-même contient l'identité de l'envoyeur, l'opération à effectuer et le « *timestamp* ». Cette étiquette de sécurité permet de faire une vérification sur l'intégrité de la requête et une authentification de l'envoyeur; la plate-forme visitée sait alors d'où provient l'agent.

Considérons à présent le scénario où un client veut acheter un billet d'avion pour un certain vol. Il crée alors un agent et l'envoie dans le réseau. L'agent va alors parcourir plusieurs hôtes de compagnies aériennes collectant des informations sur le prix du vol en question. Supposons qu'il visite les plates-formes A_1 jusqu'à A_n dans un certain ordre, et dans chaque plate-forme il utilise son programme pour interagir et demander le prix du vol de son client. Les résultats sont stockés dans son champs des données, précédemment présenté. Étant donné que l'agent contient les étiquettes de sécurité *Tag_sécurité-C* et *Tag_sécurité-S*, la plate-forme qui reçoit l'agent peut vérifier l'intégrité du code, des privilèges, et authentifier le client afin d'effectuer le contrôle d'accès. Regardons maintenant comment les données sont rajoutées à l'agent par les plates-formes. Nous notons P_i le prix proposé par la plate-forme A_i, et *D(0),* les données initiales de l'agent. Ces données consistent en un nombre aléatoire R_0 choisi par le client, et de A_1, l'identité de la prochaine plate-forme que l'agent doit visiter. En fait $D(0) = offre(0) = ([R_0, A_1]PK\text{-}C, \{hash(R_0,A_1)\}SK\text{-}C)$. R_0 et A_1 sont en fait encryptés en utilisant la clé publique du client, et la valeur de hachage appliquée sur R_0 et A_1 est signée par le client avec sa clé privée. Quand l'agent arrive à la plate-forme A_1, cette dernière rajoute son prix au champs données de l'agent sous la forme, $offre(1) = ([A_1, P_1, R_1, A_2]PK\text{-}C, \{hash(A_1, P_1, R_1, A_2, Offre(0))\}SK\text{-}A_1)$. Maintenant, *D(1)* contient *(offre(0), offre(1))*. De proche en proche, *D(i)* contient *(offre(0), ... , offre(i-1), offre(i))* et *offre(i)*

$= ([A_i, P_i, R_i, A_{i+1}]PK\text{-}C, \{hash(A_i, P_i, R_i, A_{i+1}, Offre(i-1))\}SK\text{-}A_i)$. À la fin, l'agent revient à sa plate-forme d'origine. Cette idée de chaînage des résultats est la même que celle de l'encapsulation des résultats partiels, présentée dans la section 2.3.2. Cette liaison en chaîne nous permet d'avoir l'intégrité et la confidentialité des données, la protection contre les insertions et les suppressions, et la non répudiation.

Intégrité des données

Elle est assurée par le fait que tous les résultats sont reliés et encryptés avec la clé publique du client, et leur valeur de hachage est signée par la plate-forme visitée. Ceci permet en plus d'associer chaque prix à la plate-forme qui l'a rajouté.

Confidentialité des données

Ceci est dû au fait que les prix sont encryptés avec la clé publique du client, et donc, il n'y a que lui qui peut les lire.

Protection contre les insertions

Elle est assurée par la mise en chaîne des résultats. En effet, puisque dans chaque plate-forme les offres précédentes sont incluses dans le calcul de la fonction de hachage courante, il n'est pas possible d'effectuer une fausse insertion sans que ça ne soit détecté. Cependant, il est possible qu'une plate-forme rajoute une fausse offre à la fin de la chaîne, sans que ça ne soit pour autant détecté.

Protection contre les suppressions

Il n'est pas possible de supprimer une offre pour les mêmes raisons citées dans le cas de l'intégrité des données. Cependant, il est possible pour une plate-forme i, en cas de visite ultérieure par l'agent, de supprimer toutes les offres à partir de la sienne, puis refaire une offre et l'insérer normalement.

Non répudiation

Si une compagnie aérienne fournit un prix, elle ne pourra pas le répudier par la suite, puisque elle a signé la valeur de hachage qui contient son offre.

Ce modèle est basé sur le niveau de fiabilité des *SMCs* et de leurs certificats, ainsi que sur la robustesse des fonctions cryptographiques utilisées pour protéger l'agent contre l'altération. Cependant, il n'offre pas de protection contre une plate-forme hostile dans le cas où il n'y a aucun composant fiable. En effet, dans ce cas l'intégrité des données et leur non répudiation pourront encore être assurées, mais pas l'intégrité de l'exécution même de l'agent.

2.5 Efforts de standardisation

Au moment de la rédaction de ce mémoire, il y a deux initiatives de standardisation dans le domaine des agents mobiles, à savoir, *FIPA* et *MASSIF*. D'autres informations sur d'autres initiatives peuvent être trouvées dans [OBJ00].

FIPA (Foundation for Intelligent Physical Agents) est une association à but non lucratif qui siège à Genève en Suisse, et qui a pour but de promouvoir les technologies basées sur les agents. Elle a défini une série de spécifications concernant la mobilité dans les terminaux comme les ordinateurs portables et les PDA, et dans les logiciels comme les agents mobiles. Cependant, la sécurité des agents mobiles n'a pas encore été traitée, mais il y a une spécification concernant la sécurité de la communication d'un agent avec un autre, via un manager de sécurité de la plate-forme de l'agent (*APSM*) [FIP98]. *APSM* fournit une sécurité au niveau de la couche transport, et peut protéger la communication entre les agents avec l'encapsulation des messages et des certificats à clé publique. *APSM* est aussi responsable de l'audit de sécurité ainsi que de la négociation de services de sécurité avec les *APSMs* d'autres plates-formes. La confiance inter plates-formes repose sur les certificats à clés publiques assignés au système de gestion des agents (*AMS*). Chaque plate-forme a exactement un seul *AMS*, qui est lui-même un agent (mais pas mobile) et possède une clé publique certifiée.

MASIF (Mobile Agent System Interoperability Facilities) [OBJ97] ne standardise pour le moment que les aspects de la technologie des agents mobiles qui sont nécessaires pour assurer l'interopérabilité entre les systèmes d'agents. La communication des agents est basée sur l'objet de communication *CORBA* [OBJ98]. Ainsi, les agents doivent être définis comme des objets *CORBA* qui définissent eux-mêmes leur propre sécurité. Seule la sécurité des agents mono-saut a été traitée pour l'instant. Les éléments de sécurité suivants ont alors été identifiés : nomination des agents, authentification du client, authentification mutuelle, accès à l'identité d'une plate-forme, authentification de l'agent, politiques de sécurité, intégrité, confidentialité, ré-exécution et authentification.

Nomination des agents

Ceci veut dire fournir l'information sur un agent incluant son nom, le nom de son propriétaire, son unique identité, le type de système d'agents, l'authentification du client, et l'algorithme d'authentification.

Authentification du client

Quand un client fait la requête à une plate-forme de créer un agent, il doit être authentifié avec sa carte d'identité (crédentials). Sur la base des informations de cette carte, la plate-forme détermine et applique une politique de sécurité appropriée.

Authentification mutuelle

Avant qu'un agent ne migre d'une plate-forme à une autre, les plates-formes s'échangent leurs cartes d'identité pour qu'une politique de sécurité appropriée soit appliquée avant la migration.

Accès à l'identité d'une plate-forme

Avant le transfert de l'agent, la plate-forme réceptrice peut utiliser l'interface de sécurité *CORBA* pour obtenir une référence de l'objet carte d'identité, contenant l'identité de la plate-forme émettrice.

Authentification de l'agent

Quand un agent migre entre les plates-formes, les cartes d'identité de l'agent et de la plate-forme émettrice sont envoyées à la plate-forme réceptrice. Ceci est appelé délégation composite. La plate-forme réceptrice peut choisir de limiter les droits d'accès de l'agent, si la plate-forme émettrice est moins fiable que le client qui utilise l'agent.

Politiques de sécurité

Généralement, quand une implémentation sécuritaire d'objets *CORBA* reçoit une requête, elle vérifie la carte d'identité du requérant et utilise les informations résultantes pour établir la décision du contrôle d'accès.

Intégrité, confidentialité, ré-exécution et authentification

Avec *CORBA* sécurisé, une application peut spécifier quels services de sécurité devraient être appliqués quand une opération est demandée, par exemple, le transfert d'un agent. Ces services de sécurité incluent l'intégrité, la confidentialité, la ré-exécution, ainsi que l'authentification. Les besoins en sécurité peuvent être spécifiés dans la carte d'identité du requérant, par le propriétaire de l'agent, par exemple, ou dans la référence de l'objet.

CHAPITRE III
PROTOCOLE DE SÉCURITÉ PROPOSÉ

Le protocole de sécurité que nous proposons dans ce chapitre permet de protéger l'agent mobile tout au long de sa migration et de son processus d'achat, contre les attaques de plates-formes malicieuses. Ce protocole utilise le concept d'exécution de référence comme moyen de vérification de la bonne exécution du code de l'agent. Ces exécutions de référence sont effectuées en exécutant, en parallèle avec l'agent, un clone de ce dernier dans un serveur de confiance. Pour ce faire, nous supposons que le réseau est décomposé en régions (centres d'achats par exemple) et dans chaque région, nous supposons l'existence d'un serveur de confiance (qui peut faire payer ses services). De cette façon, en plus d'utiliser d'autres mécanismes d'encryptage et de vérification, ce protocole assure l'intégrité du code de l'agent mobile et de son exécution, de ses données et de son itinéraire. En plus de sa capacité à identifier les plates-formes attaquantes, ce protocole est robuste, tolérant aux fautes et ne compromet pas l'autonomie de l'agent mobile et ses avantages par rapport au client/serveur. Dans ce chapitre, nous allons d'abord présenter les hypothèses et les entités participantes du protocole, nous décrirons ensuite plus en détail ses séquences, et finalement nous analyserons sa robustesse et sa capacité à détecter les attaques.

3.1 Hypothèses et entités participantes

Afin de bien comprendre le fonctionnement de notre protocole, nous présenterons et justifierons nos hypothèses avant de décrire plus en détail les différentes entités participantes.

3.1.1 Hypothèses

Notre première hypothèse est que le réseau est décomposé en régions et que, dans chaque région, il y a un serveur de confiance. Cette hypothèse n'est pas si exigeante

qu'elle peut en avoir l'air et peut être disponible sur Internet à différents niveaux de granularité. En effet, dans le contexte du commerce électronique, le réseau est composé de supermarchés électroniques (e-supermarchés), eux-mêmes composés d'un ensemble de magasins électroniques (e-magasins). Les e-supermarchés sont réglementés par leurs propriétaires, qui assurent la transaction entre les clients et les fournisseurs de e-magasins en fournissant l'infrastructure technologique nécessaire. Sur la base des services offerts à ses participants, le propriétaire du e-supermarché fait payer des frais d'utilisation adéquats. Parmi ces services, on peut avoir un serveur de confiance dans lequel les décisions critiques peuvent être prises. Un tel modèle est avantageux pour les clients qui ont ainsi un accès facile aux e-magasins ainsi que pour les vendeurs qui bénéficient de la visibilité du e-supermarché et ses différents services, comme le paiement électronique par exemple. Plusieurs architectures d'agents mobiles implémentant de tels modèles d'affaires ont été proposées et évaluées [PAP98] [SOH98] [VIG98]. Notre première hypothèse est donc basée sur de tels modèles et une région peut être soit un e-supermarché dans le cas d'une fine granularité, ou un ensemble de e-supermarchés dans le cas d'une granularité plus large.

Notre deuxième hypothèse est que chacune des entités participantes, à savoir l'acheteur, les sites des vendeurs et les serveurs de confiance, possède une clé publique et une clé privée. La clé publique peut servir à l'encryptage et la clé privée à la signature numérique [PFL96]. Ces paires de clés sont utilisées conformément à une infrastructure à clé publique *PKI* (Public Key Infrastructure) [ITU93], qui garantit l'association entre l'identité d'une entité et sa clé publique correspondante par l'intermédiaire d'un certificat. On suppose alors qu'à n'importe quel moment, chaque entité peut retrouver le certificat de n'importe quelle autre entité et vérifier l'intégrité et la validité de sa clé publique. De plus, nous utiliserons des fonctions de hachage irréversibles pour produire des digests compacts de messages. Par la suite, nous allons noter la clé publique d'une entité X, E_X et sa clé privée D_X. La valeur de hachage obtenue par l'application d'une fonction de hachage irréversible H à un message m sera notée hm. Plusieurs exemples de crypto-systèmes et de fonctions de hachage irréversibles peuvent être trouvés dans

[SCH96]. Cette deuxième hypothèse est indispensable dans le contexte de sécurité afin d'assurer la confidentialité, l'intégrité et la non répudiation des messages; l'infrastructure *PKI* permet d'assurer la validité de ces opérations.

Notre troisième hypothèse est que le transport des agents et les communications se font par des canaux authentifiés. Cette hypothèse correspond d'avantage à un souci d'alléger les notations des échanges de messages et de les réduire à celles qui sont strictement nécessaires pour notre protocole. En effet, cette hypothèse va nous permettre de systématiser la confidentialité par encryptage, l'intégrité et la non répudiation par hachage et signature numérique pour chaque message échangé entre deux entités, en plus de leur authentification mutuelle. Cependant, elle demeure indispensable du point de vue sécurité pour prévenir toute attaque sur les messages échangés ainsi que sur les agents mobiles pendant leur transport.

3.1.2 Entités participantes

Il y a essentiellement cinq entités principales participant à ce protocole, qui sont : la plate-forme d'origine (P_0), les plates-formes des vendeurs (P_{ij}), les serveurs de confiance (SC_j), l'agent mobile (AM) et son clone (AC). La description de chacune d'entre elles est donnée dans ce qui suit.

1. La plate-forme d'origine P_0

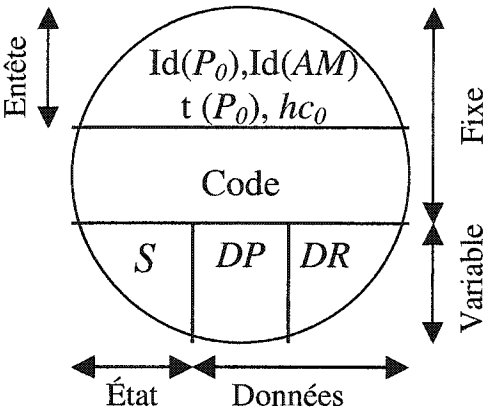
La plate-forme d'origine, notée P_0, crée l'agent mobile AM puis lui affecte un itinéraire de départ selon le produit recherché par l'utilisateur et selon ses connaissances basées sur les achats précédents. Elle crée ensuite son clone AC qui est une copie exacte de AM. Finalement, AM est lancé vers sa première plate-forme de migration P_{11} et AC vers le serveur de confiance de la région concernée, SC_1. AM va ainsi migrer d'une plate-forme P_{ij} à une autre, à la recherche du produit voulu par l'utilisateur. AC, quant à lui, migre à un autre serveur de confiance SC_j seulement quand AM change de région. Une fois la migration terminée, AM et AC reviennent à la plate-forme d'origine P_0 *avec* les résultats trouvés.

2. L'agent Mobile AM

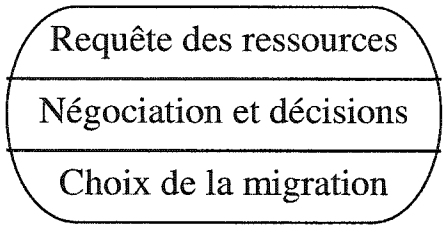
L'agent mobile *AM* est créé par P_0 selon le produit recherché par l'utilisateur. Il est composé de quatre sections principales, qui sont : son entête et son code qui représentent la partie fixe, son état et ses données qui représentent la partie variable. Ceci est illustré à la Figure 3.1a. L'entête correspond à l'information relative à l'agent et à son code et elle est sous la forme suivante : *[Id* (P_0)*, Id (AM), t* (P_0)*,* hc_0*]*. Id (P_0) désigne l'identité de la plate-forme d'origine, P_0 et Id (*AM*) celle de l'agent mobile. t (P_0) est un poinçon de temps indiquant le temps auquel *AM* a quitté P_0. hc_0 est la valeur de hachage appliquée par P_0 au code de *AM,* en utilisant la fonction de hachage *H*, i.e. $H(c)=hc_0$. L'entête est signée numériquement par la plate-forme d'origine avec sa clé privée D_{P0} pour assurer son intégrité.

Le code de l'agent mobile contient, selon son degré d'intelligence, ses préférences, son raisonnement, sa planification et son apprentissage [GIL95] (c.f. Figure 3.2). Afin de bien pouvoir illustrer la protection que procure notre protocole, le code de *AM* est composé de trois parties logiques. La première concerne la requête des ressources, la seconde, la partie relative à l'achat du produit et la dernière son itinéraire. Ceci est schématisé à la Figure 3.1b. La troisième section de *AM* est l'état *S*. Elle contient les variables d'état, i.e. les structures de données utilisées dans le programme ainsi que l'état d'exécution qui inclue la pile d'exécution et le compteur ordinal. En bref, *S* contient toute l'information nécessaire à l'agent pour continuer son exécution dans une plate-forme distante au même point où il s'est arrêté dans la plate-forme précédente. Pour ce qui est de la dernière section de *AM* qui est la section des données, elle est elle-même décomposée en deux parties : la partie des données du protocole, notée *DP,* et la partie des données résultantes, notée *DR*. La partie *DP* contient les informations sur l'itinéraire de *AM, i.e.* les plates-formes proposées à *AM*, la région à laquelle appartient chaque plate-forme et son statut, à savoir, visitée, pas visitée ou malicieuse (*M*). Cette partie contient initialement un certain nombre de plates-formes proposées par la plate-forme d'origine P_0 et elle est enrichie par la suite par les plates-formes visitées. *DP* est schématisée à la Figure 3.1c. La dernière partie de la section des données est celle

contenant les données résultantes *DR*. Ces données sont en fait les données concernant le produit recherché, recueillies dans chaque plate-forme comme son prix, sa qualité, ses garanties, etc.



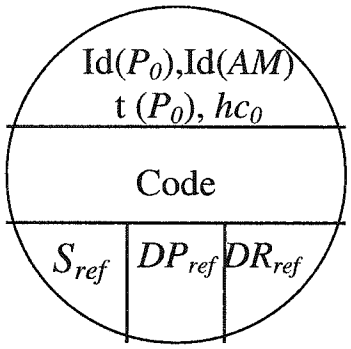
a) La composition de *AM*



b) La composition du code

Région	Plate-forme	Visitée
02	132.10.2.15	oui
06	127.20.13.4	non
46	121.50.4.63	M
15	90.13.41.62	oui
21	21.35.42.12	non

c) La partie *DP* de *AM*



d) La composition de *AC*

Figure 3.1 Composition de *AM*, de son code, de sa partie *DP*, et du clone *AC*

3. Le clone AC de l'agent mobile

L'agent cloné nommé *AC* est une copie conforme de l'agent mobile *AM*. Il contient ainsi exactement les mêmes sections que ce dernier et les mêmes informations de départ.

La seule différence est que *AC* s'exécute seulement dans les serveurs de confiance SC_j. Les informations contenues dans *AC* sont donc considérées comme intègres et de référence. Elles seront par la suite notées avec un indice *ref*, i.e. S_{ref} pour l'état, DP_{ref} pour les données du protocole et DR_{ref} pour les données résultantes. *AC* est schématisé avec ses différentes sections à la Figure 3.1d. Cet agent va servir principalement à vérifier l'exécution de *AM*, en s'exécutant en parallèle de ce dernier.

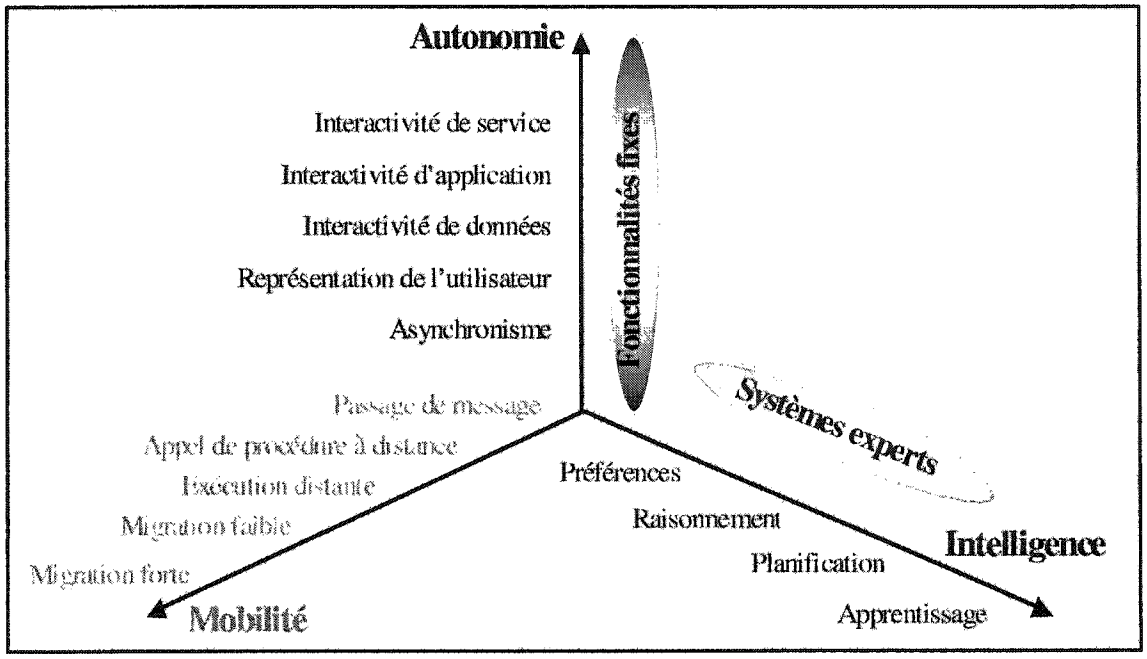


Figure 3.2 Compétences d'un agent

4. Les plates-formes des vendeurs P_{ij}

Les plates-formes des vendeurs, notées P_{ij}, sont les plates-formes que l'agent mobile *AM* visite afin de trouver le produit recherché par son utilisateur. Chaque plate-forme contient un agent stationnaire, qu'on notera AS_{ij}, avec lequel *AM* interagit. Cet agent stationnaire joue le rôle d'intermédiaire entre *AM* et l'inventaire de la plate-forme. Plus précisément, quand *AM* formule une requête, AS_{ij} interagit avec la base de données locale afin de répondre à cette requête. Ceci permet, entre autres, d'empêcher l'agent mobile d'accéder lui-même à l'inventaire du e-magasin. Cet agent stationnaire prend en

fait en charge l'agent mobile dès son arrivée à la plate-forme jusqu'à son départ. De plus, AS_{ij} s'occupe de la communication avec toutes les entités extérieures (d'autres plates-formes ou agents mobiles, etc.). En bref, il est le représentant de la plate-forme pour toutes les interactions. Comme il a été dit auparavant dans les hypothèses, le réseau est décomposé en plusieurs régions et, dans chaque région, il y a un serveur de confiance SC_j. Les P_{ij} d'une même région connaissent l'adresse du SC_j de leur région et peuvent ainsi communiquer avec lui quand c'est nécessaire. La Figure 3.3 illustre les entités participantes de notre protocole.

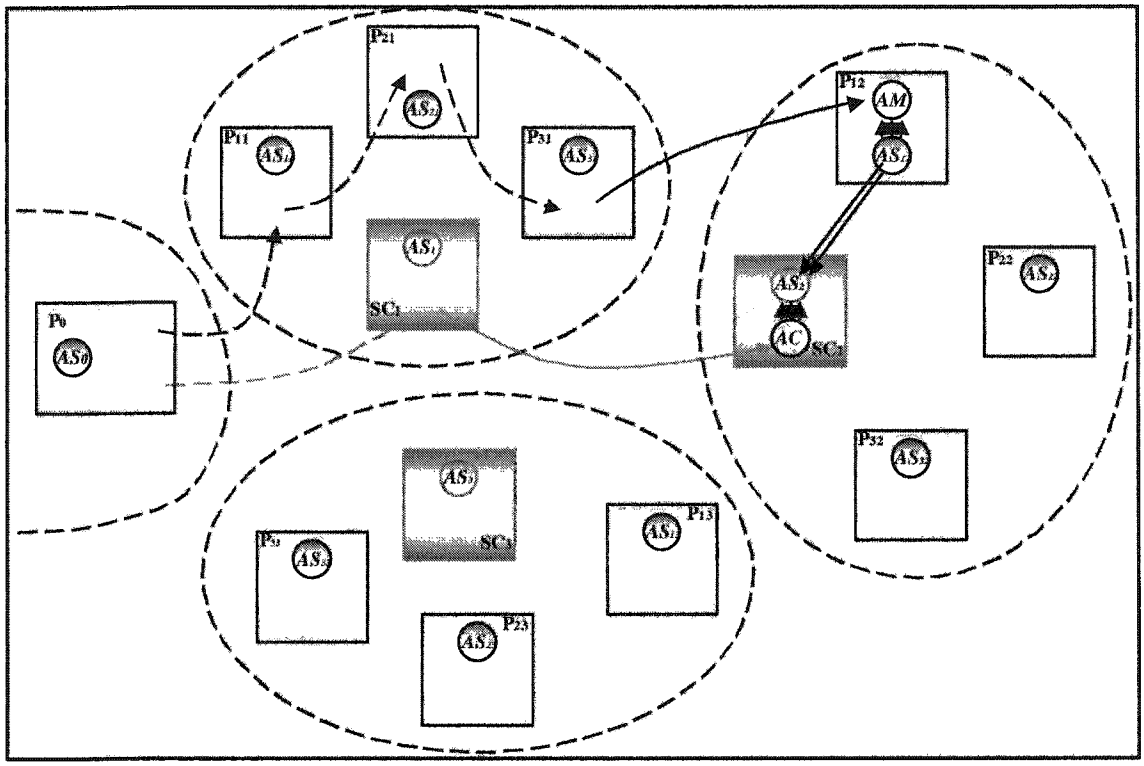


Figure 3.3 Entités participantes à notre protocole

5. Les serveurs de confiance SC_j

Les serveurs de confiance, notés SC_j, sont en fait des plates-formes qui offrent des services d'exécution et en qui le propriétaire de l'agent mobile a confiance. Comme présenté précédemment dans les hypothèses, dans chaque région du réseau il y a un serveur de confiance. Ce serveur de confiance sert principalement à exécuter le clone *AC*

grâce aux informations reçues des différentes plates-formes P_{ij} et vérifier ainsi la bonne exécution de *AM*. De la même manière que pour les autres plates-formes, SC_j est représenté par un agent stationnaire, AS_{ij}, qui s'occupe de l'exécution de *AC* et donc de ses interactions avec lui ainsi que des interactions avec les entités extérieures. De plus, chaque SC_j pourrait servir pour toute décision critique ainsi que pour effectuer la transaction d'achat, dépendamment de l'application.

3.2 Spécifications du protocole

Nous rappelons que l'agent mobile *AM* est composé d'une partie fixe et d'une partie variable, comme illustré à la Figure 3.1a. La partie fixe consiste en l'entête *[Id* (P_0)*, Id (AM), t* (P_0)*,* hc_0*]* et le code de l'agent. La partie variable consiste en l'état *S,* les données du protocole *DP* et les données résultantes *DR*. *AM* possède un itinéraire de départ, i.e. un ensemble de plates-formes proposées par l'utilisateur, à priori. Par la suite, l'itinéraire est défini dynamiquement tout au long de sa migration. Ces informations sur l'itinéraire sont stockées dans la section *DP* de *AM*, à partir de laquelle ce dernier décide de sa prochaine migration dans chaque plate-forme. Notre protocole de sécurité de l'agent mobile par un clone de référence se déroule comme suit.

Tout d'abord, l'agent stationnaire de la plate-forme d'origine P_0 crée l'agent mobile *AM* selon le ou les produits recherchés par l'utilisateur. *AM* contient alors son entête et son code, son état initial S_0 et ses données, DP_0 qui contient l'itinéraire de départ et donc la prochaine plate-forme de migration P_1 et DR_0 qui est initialement vide. Le clone *AC* de *AM* est ensuite créé. Puis, *AM* est migré vers la première plate-forme P_1 de son itinéraire et *AC* vers le serveur de confiance de la région concernée, à savoir SC_1. La composition de *AM* et de *AC,* à cette étape initiale, est représentée au Tableau 3.1.

Par la suite, *AM* migre selon son itinéraire d'une plate-forme P_{ij} à une autre pendant que *AC* est dans le serveur de confiance SC_j de la région concernée (i.e. la région à laquelle P_{ij} appartient). Le clone *AC* ne migre vers un autre serveur que quand *AM* change de région et va alors dans le serveur de confiance de cette nouvelle région. L'exécution de *AM* et de *AC* se fait ensuite selon les étapes indiquées à la Figure3.4.

Tableau 3.1 Composition des agents *AM* et *AC* à l'étape initiale

AM	*AC*
$S = S_0$	$S = (S_0)_{ref} = S_0$
$DP = DP_0$	$DP = (DP_0)_{ref} = DP_0$
$suiv_0 = P_1$	$suiv_0 = (suiv_0)_{ref} = P_1$
$DR = DR_0 = \emptyset$	$DR = (DR_0)_{ref} = \emptyset$

Quand l'agent mobile *AM* arrive dans la plate-forme P_{ij}, son agent stationnaire AS_{ij} vérifie son certificat afin de confirmer son identité, puis vérifie l'intégrité de son code grâce à la valeur de hachage hc_0 contenue dans son entête (Étape 1). Cette vérification se fait en appliquant la même fonction de hachage *H* que celle appliquée par la plate-forme d'origine P_0 à l'étape initiale. Si l'identité de *AM* n'est pas conforme à celle indiquée par le certificat, l'agent a alors été corrompu. Si la valeur de hachage trouvée par AS_{ij} est différente de hc_0, cela veut dire que le code n'est pas intègre. AS_{ij} informe alors le serveur de confiance de la région, SC_j, en indiquant dans le statut d'arrivée M_{Arr} le problème rencontré. SC_j détectera alors que $P_{(i-1)j}$, la plate-forme précédemment visitée par *AM*, a corrompu son identité ou a modifié son code. Une copie du clone est alors créée par SC_j afin de remplacer l'agent corrompu. Si les vérifications donnent les bons résultats, *AM* est alors intègre et arrive dans sa phase d'exécution dans laquelle il peut amorcer son exécution (c.f. Figure 3.4).

Au début de cette exécution (c.f. Figure 3.1b), *AM* effectue la requête des ressources dont il a besoin pour effectuer sa mission (quantité de mémoire, temps CPU, etc.). Si AS_{ij} ne reçoit pas la requête au bout d'un certain temps, elle informe le serveur SC_j du problème. Sinon, elle évalue la requête et, selon l'identité de *AM* et de son origine, et selon sa propre politique de sécurité, elle lui accorde des droits d'accès. Si ce dernier est accepté, il continue son exécution (Étape 2), i.e. il exécute la partie du code relative à la négociation du produit. Au fur et à mesure de l'exécution, l'agent

stationnaire AS_{ij} enregistre les entrées IP_i, i.e. toutes les données relatives à la négociation et que ce dernier fournit. Une fois cette partie du code terminée, il encrypte les données résultantes avec la clé publique EP_0 de la plate-forme d'origine et les stocke dans la section *DR* de l'agent mobile (Étape 3). Afin de prévenir qu'une des plates-formes suivantes puisse effacer ce résultat, nous pouvons le relier aux résultats précédents et à la l'identité de la plate-forme suivante. Ceci peut se faire en stockant le résultant DR_i sous la forme suivante : $DP_{ij}[EP_0(H(DR_{i-1}), DR_i, suiv_i)]$, $suiv_i$ étant la plate-forme suivante, trouvée par la suite de l'exécution. Par la suite, de même que dans le cas où *AM* n'a pas été accepté par AS_{ij}, ce dernier continue son exécution avec la dernière partie du code, celle relative à l'itinéraire et à la migration. L'organigramme de AS_{ij} illustrant les actions entreprises par ce dernier est présenté à la Figure 3.5. Quant à la Figure 3.6, elle illustre les actions de *AM*.

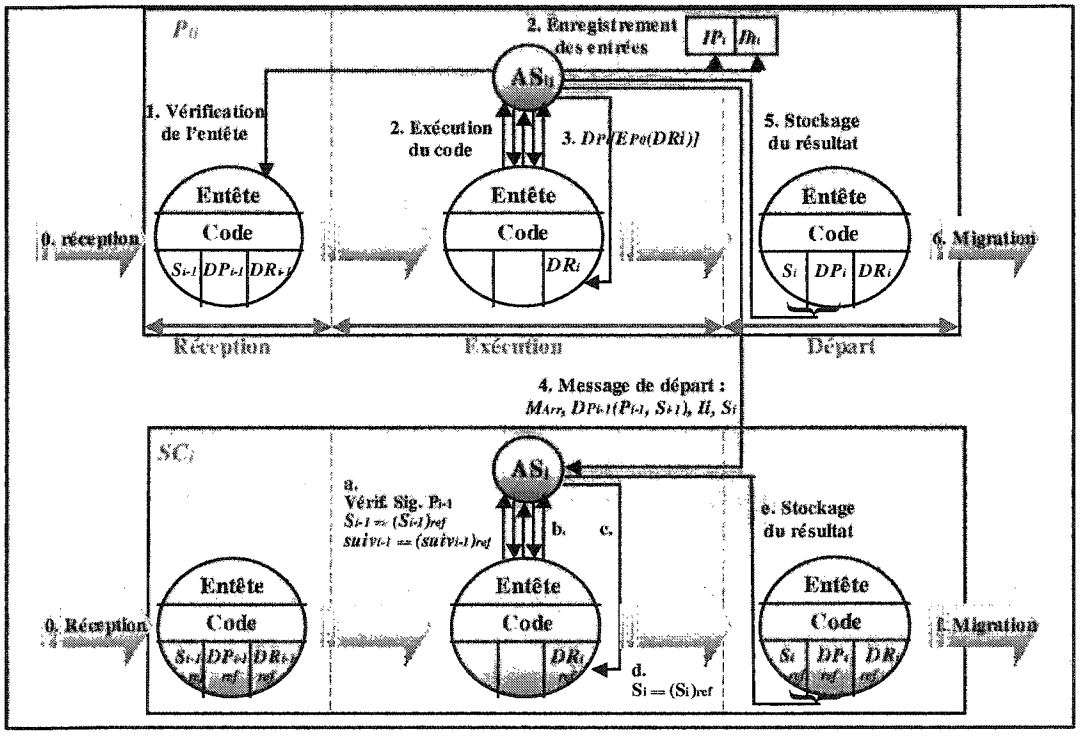


Figure 3.4 Les étapes de notre protocole

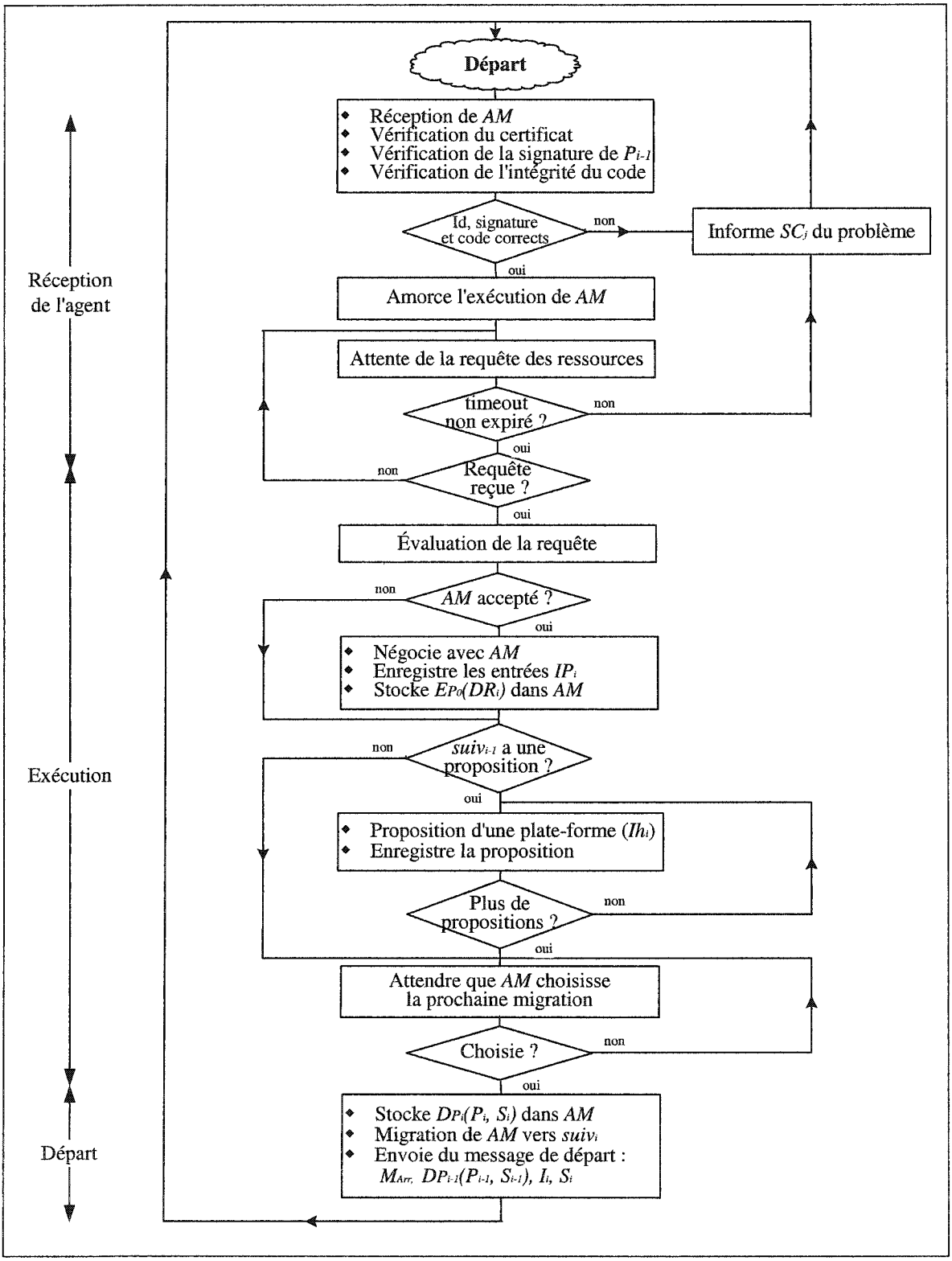


Figure 3.5 Organigramme de AS_{ij}, l'agent stationnaire de P_{ij}

Dans la partie du code relative au choix de la prochaine plate-forme, *AM* demande des recommandations à AS_{ij}, i.e. des plates-formes dans lesquelles le produit recherché pourrait être trouvé. De plus, il lui demande de proposer en priorité des plates-formes dans la région courante. Ceci correspond surtout à un souci d'optimisation de l'itinéraire et d'amélioration de la performance, qui relève normalement des préoccupations du développeur de l'application. Pour chaque proposition faite par AS_{ij}, *AM* parcourt la section *DP* et vérifie que cette proposition n'a pas déjà été faite auparavant par une autre plate-forme. Le nombre maximal des nouvelles recommandations a été fixé à trois à titre d'exemple et pourra être changé selon le produit recherché. La suite de l'exécution de cette partie concerne le choix de la prochaine plate-forme de migration.

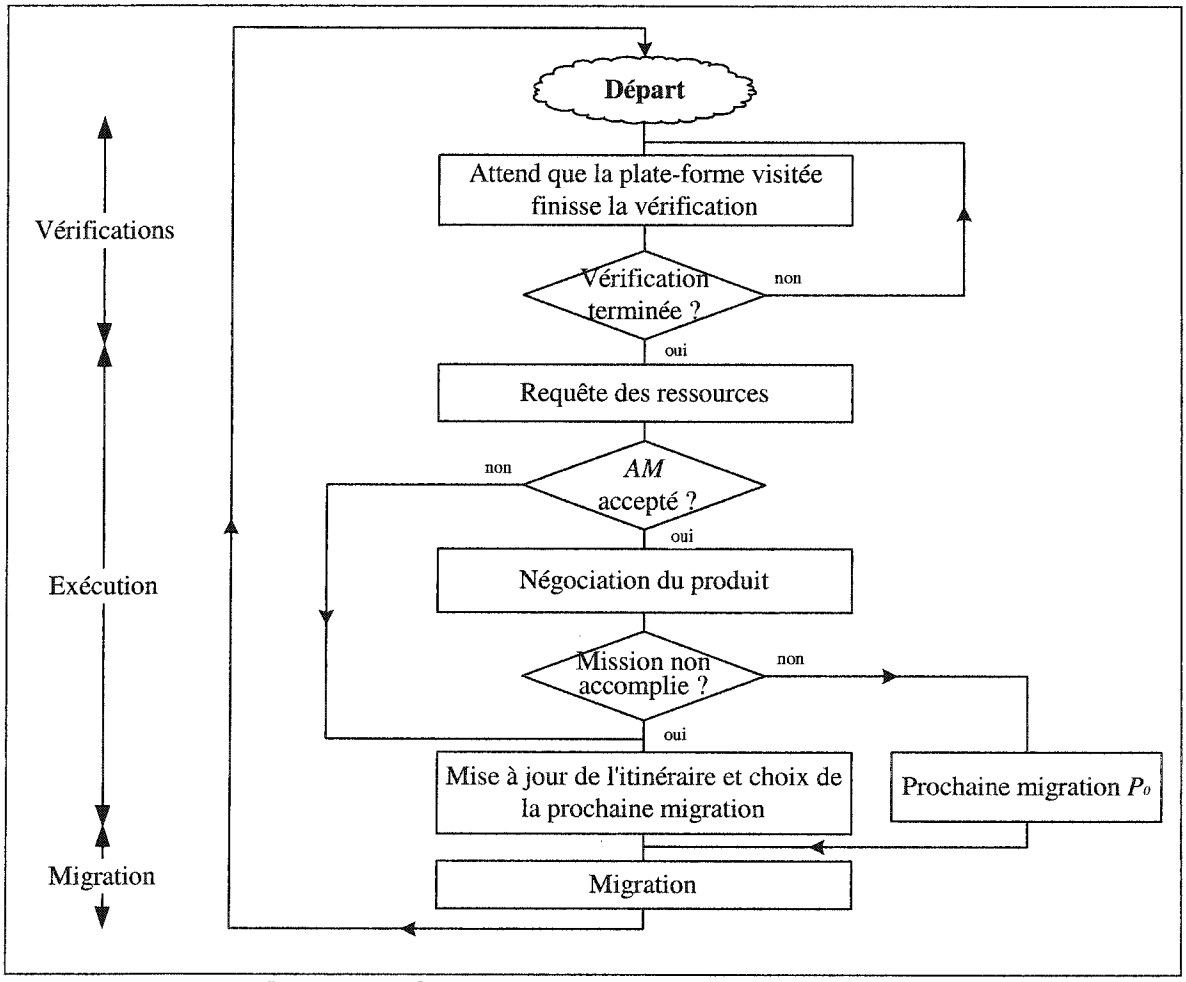


Figure 3.6 Organigramme de l'agent mobile *AM*

AM parcourt les propositions faites dans *DP* et choisit comme prochaine migration la première proposition de plate-forme qui n'a pas été visitée et en priorité dans la région courante. Au fur et à mesure de l'exécution de cette partie du code, les Ih_i représentant les propositions de plates-formes faites par AS_{ij} sont enregistrées (Étape 2). Par la suite, AS_{ij} envoie le message de départ $[M_{Arr}, DP_{i-1}(P_{i-1}, S_{i-1}), I_i, S_i]$ au serveur de confiance de la région, SC_j (Étape 4). M_{Arr} est le statut d'arrivée de l'agent qui indique si *AM* est bien arrivé chez P_{ij} et quel est le résultat des vérifications concernant son identité et l'intégrité de son code. I_i représente l'ensemble des entrées dont *AM* a eu besoin pour effectuer son exécution, i.e. les entrées IP_i relatives au produit et celles concernant la prochaine migration, Ih_i. S_i est l'état résultant. Finalement, les informations sur l'itinéraire ainsi que l'état résultant sont stockés dans l'agent mobile *AM* (Étape 5) et ce dernier migre vers la prochaine plate-forme (Étape 6). À la Figure 3.4 sont illustrées ces différentes étapes et à la Figure 3.7 est illustré l'organigramme détaillant les étapes effectuées par *AM* pour choisir sa prochaine migration.

Comme il a été dit auparavant, le serveur de confiance SC_j sert à vérifier l'exécution de *AM* dans les différentes plates-formes en utilisant son clone *AC*. Une fois le clone reçu, l'agent stationnaire AS_j du serveur de confiance vérifie si le message indiquant les données d'exécution a été reçu. Ce message contient les informations suivante : M_{Arr}, $D_{Pi-1}(P_{i-1}, S_{i-1})$, I_i et S_i. Quand ce message est reçu, AS_j vérifie d'abord si *AM* a migré au bon endroit en vérifiant l'égalité $suiv_{i-1}==(suiv_{i-1})_{ref} = P_{ij}$ (Étape a). Si ce n'est pas le cas, le coupable est $P_{(i-1)j}$ pour ne pas avoir envoyé *AM* au bon endroit. AS_j avorte alors l'exécution de *AM* et envoie une copie du clone à la place de l'agent. Cette procédure est expliquée plus en détail par la suite. Si par contre *AM* a migré au bon endroit, AS_j vérifie alors si la signature sur l'état d'entrée dans le message $D_{Pi-1}(P_{i-1}, S_{i-1})$ est correcte, i.e. que P_i a bien exécuté l'agent avec l'état reçu de la part de $P_{(i-1)j}$ (Étape a). Si ce n'est pas le cas, cela veut dire que P_{ij} a envoyé une mauvaise signature et est désigné comme coupable. La procédure d'avortement est alors exécutée. P_{ij} est coupable dans ce cas car si c'était $P_{(i-1)j}$ qui avait envoyé une fausse signature, P_{ij} l'aurait alors signalé lors de la réception.

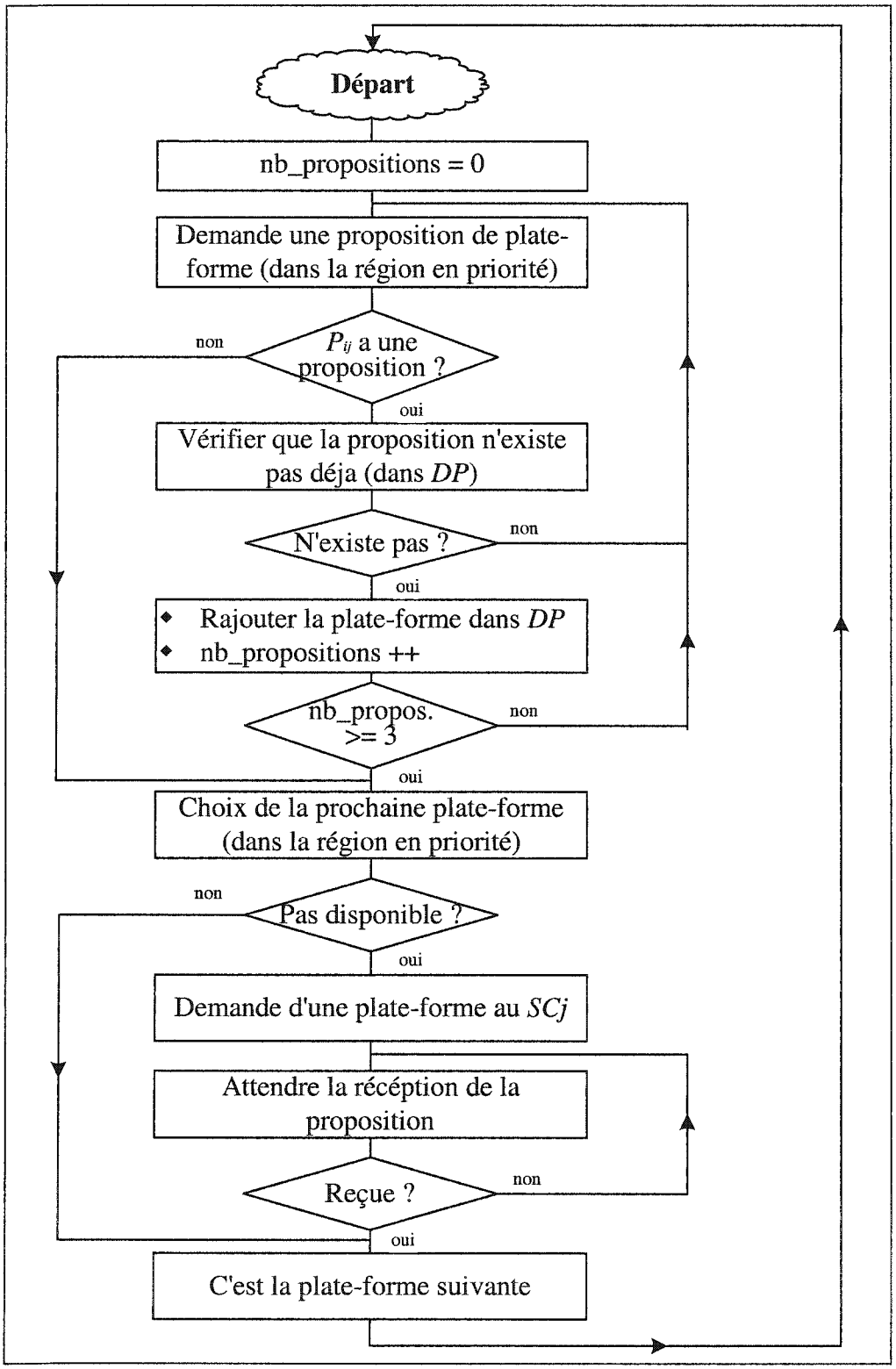


Figure 3.7 Organigramme du choix de la prochaine migration par *AM*

Par la suite, l'égalité $S_{i-1}==(S_{i-1})_{ref}$ est vérifiée (Étape a). Si elle est fausse cela veut dire que soit l'agent a été avorté à l'étape précédente, i.e. lors de la vérification de l'état résultant de l'exécution dans $P_{(i-1)j}$, et cette dernière aura été désignée coupable. Soit que $P_{(i-1)j}$ a exécuté l'agent correctement, mais a volontairement envoyé un résultat erroné à P_{ij}. Dans ce cas aussi, $P_{(i-1)}$ est désignée coupable, étant donné qu'elle a signé cet état et ne pourra pas donc le nier. À ce niveau de vérification, AS_j sait que P_{ij} a bien exécuté l'agent avec les informations d'état reçus de $P_{(i-1)j}$. Ce dernier exécute alors le clone avec les données d'entrées I_i reçues (Étape b). Il exécute alors la partie du code relative à la négociation lui permettant de trouver le prix conclu, puis celle du choix de la migration lui permettant de savoir où est ce que *AM* devrait être (c.f. Figure 3.1b). Les données résultantes sont stockées dans le clone après encryptage et signature numérique (Étape c). À l'issue de l'exécution, AS_j connaît l'état résultant $(S_i)_{ref}$. Il procède alors à la vérification de la validité de l'état résultant en vérifiant l'égalité $S_i==(S_i)_{\text{ref}}$ (Étape d). Si ce n'est pas le cas, P_{ij} est désignée coupable pour avoir mal exécuté le code de l'agent et la procédure d'avortement est alors lancée. Si par contre l'égalité est vérifiée, AS_j sait alors que P_{ij} a bien exécuté *AM*, et les données résultantes ainsi que le nouvel état sont stockés dans le clone (Étape e). Finalement, si la prochaine plate-forme de migration de *AM*, ou de la copie du clone dans le cas d'une attaque, n'est pas dans la même région que P_{ij}, le clone *AC* ou une copie de ce dernier, selon le cas, est migré au serveur de confiance de cette région (Étape f), sinon il continue les vérifications pour l'exécution suivante. La Figure 3.8 illustre l'organigramme de AS_j présentant les différentes étapes citées. Par ailleurs, le Tableau 3.2 présente la composition de *AM* et *AC* à l'étape *i* de la vérification.

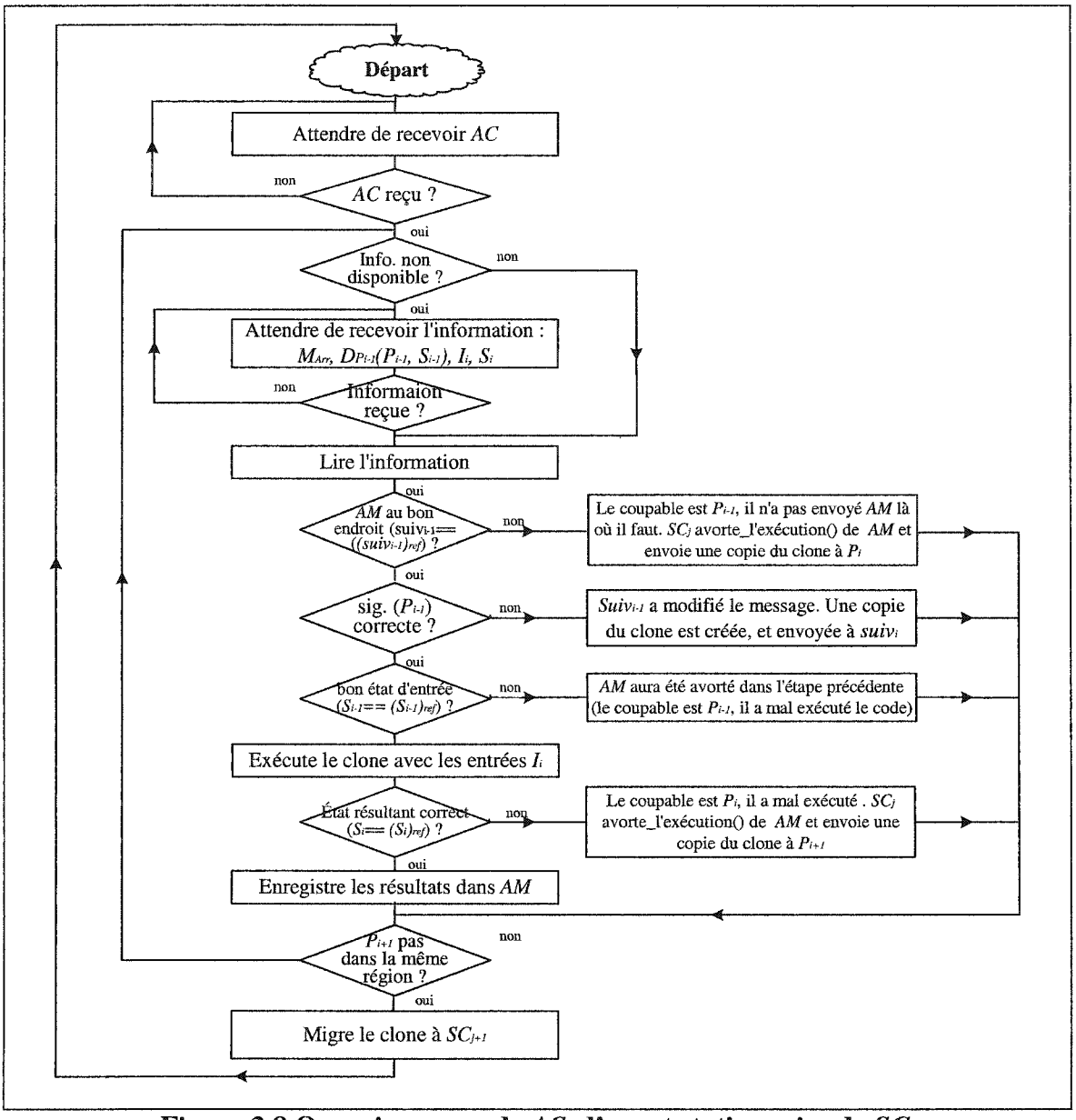


Figure 3.8 Organigramme de AS_j, l'agent stationnaire de SC_j

Tableau 3.2 Composition des agents *AM* et *AC* à l'étape *i* de la vérification

AM	*AC*
$S = S_{i+k}$	$S = (S_i)_{ref} = S_i$
$DP = DP_{i+k}$	$DP = (DP_i)_{ref} = DP_i$
$Suiv_{i+k} = P_{i+k+1}$	$Suiv_i = (suiv_i)_{ref} = P_{i+1}$
$DR = D_{P(i+k)j}[EP_0(H(DR_{i+k-1}), DR_{i+k}, suiv_{i+k+1})]$	$DR = D_{SCj}[EP_0(H(DR_{i-1}), DR_i, suiv_i)]$

Nous allons maintenant détailler le fonctionnement de la méthode avorte_exécution(), que SC_j exécute quand l'agent n'est pas dans la plate-forme où il devrait être, ou quand l'état résultant de l'agent n'est pas correct. Cette méthode sert, comme son nom l'indique, à avorter l'exécution de l'agent *AM* quand une des vérifications a échoué, indiquant par conséquent qu'il a été compromis. Le problème qui se pose alors est de pouvoir localiser l'agent, étant donné que sa migration peut se faire plus rapidement que la vérification faite par le serveur de confiance. En d'autres termes, il peut arriver que SC_j soit en train de vérifier l'exécution dans P_{ij} alors que *AM* est déjà en train de s'exécuter dans la plate-forme P_{i+k}, k étant un entier. De plus, *AM* peut avoir changé de région entre temps. Cependant, SC_j peut quand même savoir où est ce que l'agent est en train de s'exécuter, étant donné qu'il reçoit le message de la plate-forme qui l'exécute. La localisation et l'avortement de l'exécution de l'agent se fait alors comme présenté à la Figure 3.9.

Tout d'abord, SC_j parcourt tous les messages reçus jusqu'au dernier. Si ce message indique que l'agent a changé de région, SC_j contacte alors le serveur de confiance de la région concernée et lui demande d'avorter l'exécution de l'agent *AM*. Ce dernier exécute à son tour la méthode avorte_exécution() afin de localiser l'agent puis l'avorter. Si par contre le dernier message reçu indique que *AM* est dans la même région, SC_j avorte à distance l'exécution de *AM* et attend la confirmation de cet avortement. Si l'avortement n'a pas réussi, cela veut dire que l'agent était en mouvement lorsque SC_j essayait de l'avorter et la procédure d'avortement est alors recommencée depuis le

début. Si par contre l'avortement est confirmé, SC_j envoie aux plates-formes ayant exécuté l'agent corrompu un message d'annulation. À ce moment là, SC_j exécute le clone AC en marquant la plate-forme fautive comme étant malicieuse et stocke cette information dans la section DP du clone. Ce dernier est ensuite exécuté pour déterminer la prochaine migration. Finalement, une copie du clone qui sera le nouvel agent est créé et migre vers la prochaine plate-forme de l'itinéraire.

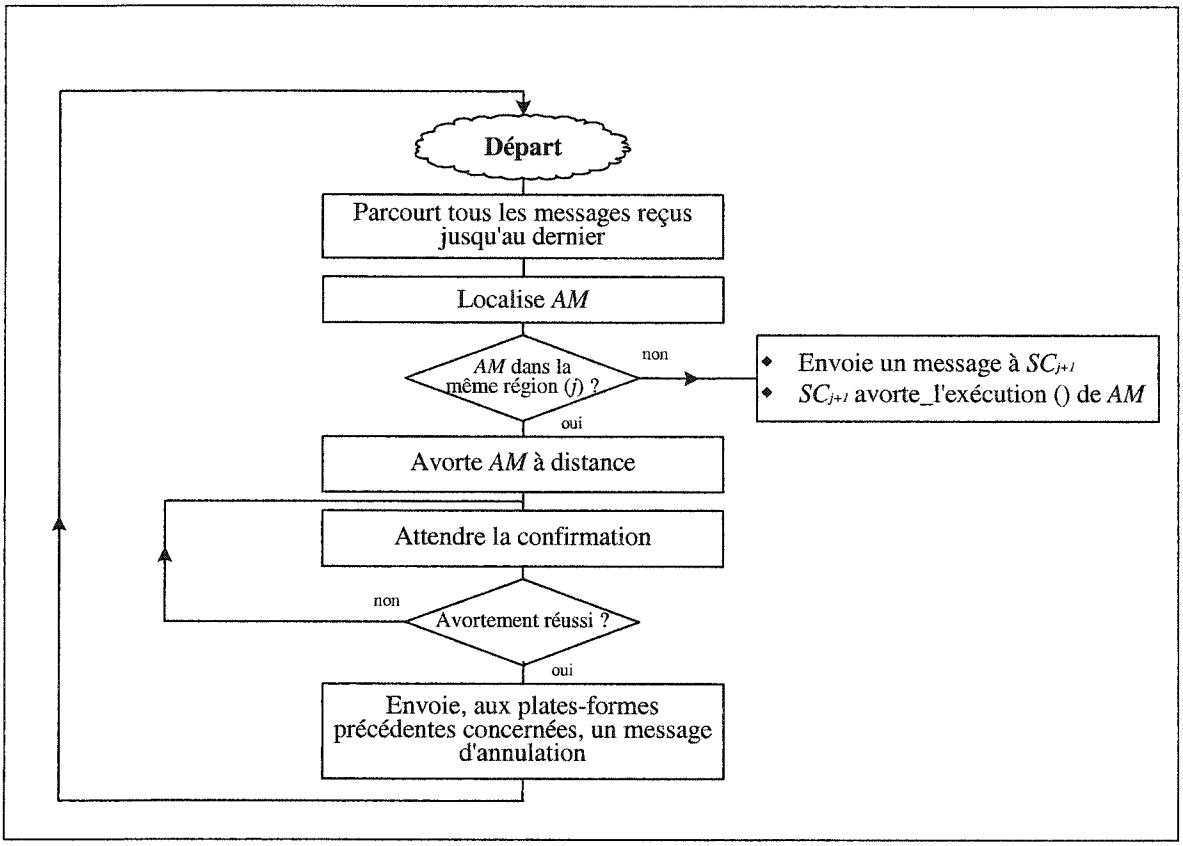


Figure 3.9 Organigramme de avorte_exécution(), exécuté par AS_j

3.3 Sécurité du protocole

Après avoir décrit les fonctionnalités de notre protocole ainsi que ses séquences, nous allons dans cette partie présenter la manière dont ce dernier permet de détecter différents types d'attaques et analyser son efficacité à désigner le coupable. Pour ce faire, nous allons passer en revue toutes les attaques décrites dans le chapitre précédent

et démontrer comment ces dernières sont détectées par notre protocole et par quels mécanismes.

3.3.1 Mascarade

Rappelons, comme défini au chapitre 2, qu'une mascarade est le fait de donner de fausses informations à l'agent, soit en lui faisant croire qu'il est au bon endroit alors que ce n'est pas le cas, soit en l'envoyant vers une mauvaise destination. Une autre forme de mascarade est de créer un clone et le faire passer pour l'agent original auprès des autres plates-formes.

Donner une fausse identité à AM

Dans notre protocole, l'identité de la plate-forme P_{ij} accueillant l'agent mobile est envoyée au serveur de confiance SC_j au début de la session de communication dans laquelle P_{ij} envoie les informations d'exécution de *AM*. D'après l'hypothèse 3, P_{ij} ne peut pas envoyer une mauvaise identité car les communications se font par des canaux authentifiés, dans lesquels les identités sont échangées mutuellement. Chaque plate-forme s'authentifie en utilisant sa clé privée. Changer d'identité signifie alors que P_{ij} utilise la clé privée d'une autre plate-forme, ce qui est impossible à moins que cette dernière soit complice. Dans ce cas, cette plate-forme sera tenue responsable si une attaque se produit.

Envoyer AM vers une fausse destination

Après avoir fini d'exécuter *AM*, P_{ij} peut par contre le forcer à migrer vers une autre plate-forme que celle où il voulait aller. Ceci sera détecté par le serveur de confiance SC_j qui s'en rendra compte lors des messages échangés. En effet, quand *AM* va arriver à la mauvaise destination, il va envoyer un message à SC_j. Ce dernier apprendra ainsi l'identité de cette plate-forme car la communication est authentifiée (hypothèse 3) et se rendra compte que c'est la mauvaise destination, en la comparant avec celle trouvée par le clone. La mauvaise destination ne pourra pas se faire passer pour la bonne, car elle ne

pourra pas forger la clé privée de la bonne plate-forme pour effectuer une bonne authentification.

Remplacer AM par un clone

La plate-forme P_{ij} peut décider de cloner l'agent, modifier ses fonctionnalités et le faire passer pour l'agent original. Ceci peut se faire soit en gardant l'entête de ce dernier ou en la modifiant. Dans les deux cas, SC_j va détecter l'attaque lors des échanges des messages avec la plate-forme accueillant l'agent cloné. Ce cas d'attaque est détaillé par la suite, étant donné qu'il correspond à l'altération de l'agent.

3.3.2 Déni de service

Cette attaque peut se manifester selon plusieurs formes. En recevant l'agent *AM*, la plate-forme P_{ij} peut ignorer sa requête de ressources et lui en donner moins que ce dont il a besoin, entravant ainsi son exécution. Elle peut aussi mal exécuter son code ou ne pas l'exécuter du tout, ou encore, terminer ou tuer l'agent sans notification. Finalement, elle peut aussi introduire volontairement des délais inacceptables pour des transactions critiques telles que l'achat d'actions. Ces types d'attaques sont les plus difficiles à détecter, étant donné qu'elles ne se manifestent pas forcément par une action malicieuse.

Mauvaise exécution du code

Considérons le cas de la mauvaise exécution du code. Le cas de la mascarade et d'altération étant traités à part dans d'autres sections, nous allons considérer que la plate-forme envoie bien l'état résultant effectivement trouvé. Si le code de l'agent est mal exécuté, SC_j s'en rendra compte lors de la vérification de l'état résultant. En effet, en exécutant le clone avec les I_i reçus, SC_j trouve l'état de référence résultant et compare ce dernier à l'état reçu de la plate-forme et saura si celle-ci a mal exécuté le code.

Terminaison de AM sans notification

Pour ce qui est de la terminaison de l'agent sans notification, là encore ce sera détecté par SC_j. Ce dernier sait à chaque étape où est ce que l'agent se trouve grâce aux messages qu'il reçoit et à l'exécution du clone. En effet, après avoir exécuté l'agent, chaque plate-forme envoie un message attestant le départ de ce dernier, que SC_j utilise pour trouver quelle est sa prochaine migration. Ainsi, si SC_j ne reçoit pas de message au bout d'un certain temps, il saura quelle plate-forme a mal agi. Cependant, si une plate-forme P_{ij} envoie le bon message à SC_j mais n'envoie pas vraiment l'agent à $P_{(i+1)j}$, cette dernière va être accusée à tort, étant donné que SC_j va croire que l'agent se trouve chez elle alors que ce n'est pas le cas. Pour remédier à cela, il faut utiliser le protocole de non répudiation avec un serveur de confiance pour transférer les agents, pour que P_{ij} ait la preuve que $P_{(i+1)j}$ a bien reçu l'agent. De plus, P_{ij} doit joindre cette preuve au message qu'elle envoie à SC_j. À ce moment là, si P_{ij} n'a pas de preuve qu'elle a envoyé l'agent, c'est elle qui sera tenue responsable de la perte de l'agent et à juste titre. Ceci relève cependant de la façon dont la migration a été implémentée, ce qui est au-delà des objectifs de ce mémoire.

AM est gardé plus longtemps que prévu

Pour ce qui est de l'attaque où P_{ij} ignore la requête des ressources de *AM* ou le garde volontairement plus longtemps que prévu, notre protocole ne permet pas de détecter cette attaque. En effet, nous n'avons pas de moyen de savoir si l'agent prend plus de temps que prévu à cause de la charge actuelle de la plate-forme, ou bien cette dernière introduit des délais volontairement. Une possibilité pour pallier cette attaque serait de demander à la plate-forme d'envoyer au serveur de confiance une estimation du temps qu'elle compte garder l'agent, selon sa charge actuelle et la tâche demandée par ce dernier. Par ailleurs, si la plate-forme prend plus de temps que celui estimé, elle envoie un message indiquant le temps supplémentaire ainsi que la cause de ce prolongement. De cette façon, le serveur de confiance peut mieux prédire les

déplacements de l'agent, et surtout, si ce dernier est gardé abusivement par la plate-forme.

3.3.3 Espionnage

Ce type d'attaque concerne l'espionnage de l'information sensible de l'agent ainsi que ses communications avec des entités extérieures. Étant donné que la plate-forme accueillant l'agent a le contrôle total sur ce dernier, elle peut très bien espionner son code, son exécution et ses communications extérieures. De ce fait, nous ne pouvons pas prévenir ce type d'action. Cependant, notre protocole permettrait de détecter toute conséquence de cet espionnage résultant en une des autres attaques décrites dans ce mémoire.

Accès non autorisé aux données recueillies

Pour ce qui est de l'espionnage des données recueillies par l'agent dans les autres plates-formes, notre protocole permet de les prévenir et de les détecter dans le cas échéant. En effet, le prix résultant de la négociation dans chaque plate-forme est encrypté avec la clé publique de la plate-forme d'origine et signé numériquement avec la clé privée de la plate-forme ayant exécuté l'agent. De ce fait, les plates-formes visitées par la suite ne peuvent pas espionner les propositions faites par les plates-formes précédentes.

3.3.4 Altération

Cette attaque consiste en la modification des informations contenues dans l'agent, la modification des différentes composantes du message que ce dernier communique au serveur de confiance, ainsi que l'altération des données recueillies dans chaque plate-forme.

Modification du code de AM

Si une plate-forme modifie le code de l'agent en gardant la même entête, ceci sera détecté par la plate-forme suivante dès sa réception, grâce à la valeur de hachage hc_0 contenue dans l'entête, qui est signée par la plate-forme d'origine. En effet, avant d'exécuter le code, la plate-forme vérifie son intégrité en appliquant la même fonction de hachage utilisée par la plate-forme d'origine. Si la valeur trouvée est égale à hc_0 le code est alors intègre. Si par contre l'entête est aussi modifiée pour correspondre à la modification du code, ceci sera détecté aussi, étant donné que la plate-forme attaquante ne pourra pas forger la clé privée de la plate-forme d'origine pour signer l'entête. Dans ces deux cas, la plate-forme détectant l'attaque la signale au serveur de confiance qui désigne le coupable et prend les mesures nécessaires pour relancer le protocole.

Modification de l'état d'entrée ou de la signature sur cette dernière

Quand P_{ij} finit d'exécuter l'agent mobile *AM*, le message $[M_{Arr}, DP_{i-1}(P_{i-1}, S_{i-1}), I_i, S_i]$ est envoyé au serveur de confiance de la région SC_j. Si P_{ij} est malicieuse, elle peut décider soit de ne pas envoyer ce message, ou encore d'envoyer des informations erronées dans ce dernier. Si elle n'envoie pas du tout de message, au bout d'un certain temps d'attente, SC_j va détecter le comportement malicieux de cette dernière comme expliqué précédemment. La plate-forme P_{ij} peut par contre envoyer à SC_j un faux message en modifiant l'une de ses composantes $DP_{i-1}(P_{i-1}, S_{i-1})$, I_i ou S_i. L'altération de l'état d'entrée peut arriver dans deux cas. Soit P_{ij} envoie effectivement l'état S_{i-1} reçu de $P_{(i-1)j}$ mais déjà erroné et cette dernière aura déjà été désignée coupable à l'étape précédente ou va l'être à cette étape. Soit P_{ij} envoie exprès un mauvais S_{i-1}, en faisant croire que c'est celui qui lui a été envoyé par $P_{(i-1)j}$, et pour que cette dernière soit accusée à tort. Ceci sera aussi détecté, étant donné que P_{ij} ne peut pas forger la signature de $P_{(i-1)j}$ pour former le message $DP_{i-1}(P_{i-1}, S_{i-1})$.

Modification des entrées Ii et de l'état de sortie

Si P_{ij} décide d'envoyer de mauvaises informations d'entrée I_i, ceci sera détecté aussi. Rappelons que I_i sont les informations que P_{ij} fournit à *AM* lors de son exécution. Ces entrées incluent IP_i, relatives à la négociation du produit et, Ih_i, qui sont les propositions de plates-formes faites par P_{ij}. Il n'est pas dans l'intérêt de P_{ij} d'envoyer de faux IP_i, étant donné que ces derniers concernent la négociation du produit et serviront de preuves de négociation avec la plate-forme. Cette dernière sera donc obligée de se tenir à ces engagements de vendre au résultat de la négociation, si jamais *AM* choisit de faire l'achat chez-elle. Rappelons aussi que les I_i vont influencer directement l'état résultant S_i et donc une modification des I_i va engendrer une modification de S_i. De ce fait, si P_{ij} envoie des I_i qui ne sont pas cohérents avec S_i, cette attaque sera détectée et P_{ij} désignée comme coupable. En effet, l'état de référence généré avec les I_i envoyés en exécutant le clone ne va pas être égal à l'état S_i reçu dans le message.

Si par contre P_{ij} envoie des I_i autres que ceux qu'il a utilisés mais envoie l'état résultant correspondant, SC_j ne va pas s'en rendre compte, étant donné que sa vérification va être correcte. Deux cas se présentent cependant. Si P_{ij} envoie par la suite à $P_{(i+1)j}$ le bon S_i comme état d'entrée, lors de la vérification, SC_j va détecter que l'état reçu de la part de $P_{(i+1)j}$ n'est pas celui qui a résulté à l'étape précédente. Vu que les deux messages auront été signés par P_{ij}, ce dernier ne peut pas nier les avoir envoyés et est alors désigné coupable. Si par contre P_{ij} envoie à $P_{(i+1)j}$ le même état que celui qui a été envoyé à SC_j, ce dernier ne va pas le détecter malgré le fait qu'il soit faux. Cependant, P_{ij} sera obligé de tenir ses engagements concernant la négociation effectuée et dont l'information se trouve dans les IP_i envoyés.

Modification des données résultantes

Pour ce qui est de l'altération des données, si une plate-forme modifie un des résultats précédemment recueillis par l'agent, cette dernière sera obligée de fournir une signature. Lorsque l'agent revient à la plate-forme d'origine, celle-ci se rendra compte de cette modification, étant donné que la signature ne va pas correspondre à l'identité de

la plate-forme ayant exécuté l'agent. De plus, une plate-forme ne pourra pas enlever le prix proposé par une autre plate-forme sans être détecté, étant donné que chaque prix est relié au prix précédemment recueilli ainsi qu'à l'identité de la plate-forme suivante.

Pour toutes les attaques détectées par notre protocole compromettant l'agent mobile, des mesures de recouvrement sont déclenchées par le serveur de confiance ayant détecté ces attaques. En effet, ce dernier avorte l'exécution de l'agent compromis à distance, crée un nouvel agent à partir du clone de référence, et envoie finalement ce dernier à la plate-forme suivant la plate-fore malicieuse dans l'itinéraire de l'agent initial. De ce fait, l'agent peut continuer sa mission et indiquer l'identité de la plate-forme malicieuse à la plate-forme d'origine, une fois revenu. Cette dernière peut mettre l'identité de cette plate-forme dans une liste noire qu'elle publie sur Internet afin d'éviter de la visiter dans les migrations suivantes.

3.4 Analyse du protocole

Notre protocole présente plusieurs avantages de par sa capacité de détection d'une large gamme d'attaques, sa robustesse et sa tolérance aux fautes. De plus, il ne surcharge pas l'agent par des informations de contrôle et ne compromet son autonomie et l'avantage qu'il procure par rapport au client/serveur. Cependant, il recèle aussi des inconvénients étant donné l'approche utilisée. En effet, le fait d'utiliser des serveurs de confiance constitue une contrainte pour la mise en œuvre de notre protocole, qui devient d'autant plus contraignante si on veut le déployer à grande échelle.

3.4.1 Avantages

Contrairement à d'autres approches utilisées dans la littérature, notre protocole permet de faire la vérification de l'exécution de l'agent mobile en temps réel et nous offre la garantie de l'authenticité des résultats trouvés. En effet, la vérification de l'exécution de l'agent mobile se fait au fur et à mesure de son exécution et permet donc de détecter l'attaque presque aussitôt qu'elle se produit tout en désignant le coupable. De plus, le mécanisme de recouvrement de notre protocole permet à l'agent de continuer sa

mission en conservant les résultats recueillis avant l'attaque, grâce au clone de référence utilisé. Ceci n'est pas le cas dans une approche telle que les traces cryptographiques [VIG98b], dans laquelle la taille de l'agent s'agrandit d'une migration à une autre avec les traces d'exécution. De plus, dans une telle approche, l'attaque ne pourra être détectée que lorsque l'agent revient à sa plate-forme d'origine et tous les résultats suivants l'attaque seront compromis. En outre, si l'agent est kidnappé pendant son exécution, toutes les données recueillies par ce dernier seront perdues et l'utilisateur ne s'en rendra compte que plus tard.

Par ailleurs notre protocole est robuste et tolérant aux fautes et fonctionne pour des applications multi-agents. Il est robuste car il permet de détecter efficacement les attaques en temps réel sans compromettre l'avantage de l'agent mobile par rapport au client/serveur. Il est tolérant aux fautes car les données recueillies ne sont pas perdues à la suite d'une attaque, contrairement à beaucoup de méthodes décrites dans la littérature, comme nous l'avons dit précédemment. De plus, en cas de panne d'un serveur de confiance, il n'y a que les données relatives à l'exécution de l'agent dans la région concernée qui sont perdues. À ce moment là, l'exécution peut reprendre à la dernière plate-forme visitée dans la région précédente. En outre, notre protocole fonctionne pour des applications multi-agents grâce à la disponibilité de plusieurs serveurs de confiance. En effet, nous pouvons très bien imaginer une application utilisant un agent dans chaque région qui est sécurisé par le serveur de confiance de cette dernière. Quand l'un d'eux trouve le prix recherché, il informe les autres afin qu'ils arrêtent les recherches, et qu'ils reviennent éventuellement à la plate-forme d'origine.

Finalement, le fait d'utiliser plusieurs serveurs de confiance sert non seulement à rendre notre protocole plus robuste et plus tolérant aux fautes et aux pannes mais sert en plus à contenir le trafic généré dans la même région. En effet, chaque plate-forme communique seulement avec des plates-formes proches géographiquement d'elles, ce qui permet de générer du trafic seulement localement, qui se dissipe rapidement et contribue aussi à diminuer le temps d'exécution. De plus, l'utilisation de plusieurs serveurs de confiance évite aussi de constituer un goulot d'étranglement avec un seul

serveur. Dans les approches citées dans la littérature utilisant un serveur de confiance [COR99], toutes les plates-formes visitées communiquent avec le même serveur de confiance, même si elles sont éloignées géographiquement de ce dernier. En plus de constituer un goulot d'étranglement, de telles approches génèrent du trafic qui transite dans tout le réseau augmentant les temps de transmission et affectant les autres communications.

3.4.2 Inconvénients

L'inconvénient majeur de notre protocole est la supposition de l'existence d'un serveur de confiance dans chaque région du réseau. Cette hypothèse suppose l'utilisation d'un modèle de supermarchés électroniques qui, pour l'instant, n'a pas de standard établi. De plus, beaucoup de questions restent ouvertes quant à la gestion de ces serveurs de confiance, leurs confiances mutuelles, leur disponibilité à grande échelle, et le coût supplémentaire de l'utilisation de leurs services. Par ailleurs, la capacité de traitement de chaque serveur de confiance et sa gestion de la sécurité des différents agents s'exécutant dans ce dernier, dépendent des mécanismes de sécurité utilisés. Toutes ces questions dépendent des travaux de recherche en cours dans ce domaine et du standard qui pourra être adopté par les vendeurs.

L'utilisation de ces serveurs de confiance engendre un autre inconvénient qui est relié à l'optimisation de l'itinéraire de l'agent et la performance engendrée. En effet, étant donné que ce dernier est établi dynamiquement et que le clone migre vers un autre serveur de confiance à chaque fois que l'agent change de région, on peut avoir des cas où le clone se déplace à chaque déplacement de l'agent dégradant ainsi la performance. Ceci peut arriver dans des scénarios où les plates-formes proposées ne sont pas ordonnées par région. Dans de tels cas, l'agent visite une plate-forme dans la région puis migre vers une autre plate-forme dans une autre région pour revenir éventuellement dans la première région. Dans ce cas, le clone change de région à chaque migration de l'agent. Afin de remédier à ce phénomène, nous avons introduit une contrainte aux plates-formes visitées afin qu'elles proposent des plates-formes dans la région courante

en priorité. De même, pour son choix de la prochaine migration, l'agent mobile choisit de migrer vers une plate-forme dans la même région en priorité.

Finalement, une dernière faiblesse de notre protocole réside dans le fait que l'exécution du clone n'est pas synchronisée avec celle de l'agent. De ce fait, avant qu'une attaque ne soit détectée, l'agent compromis aura déjà effectué des exécutions supplémentaires et c'est seulement par la suite qu'il est avorté. Toutefois, les plates-formes recevant l'agent compromis peuvent dans un premier temps vérifier l'intégrité de son code et lui attribuer des droits d'accès selon la confiance faite à sa plate-forme d'origine. De ce fait, tout ce que ces plates-formes risquent est d'avoir utilisé des ressources pour une exécution qui sera annulée. Nous avons pensé au début à synchroniser les mouvements de l'agent mobile avec son clone, pour que ce dernier migre seulement après que le serveur de confiance ait assuré qu'il n'a pas été compromis. Nous avons laissé tomber cette approche car elle engendre un surcoût en temps d'exécution. Nous avons alors minimisé l'impact de l'absence de la synchronisation dans notre protocole en arguant qu'une plate-forme connaissant ses capacités de détection ne va pas s'attaquer à l'agent en sachant qu'elle sera détectée par la suite.

CHAPITRE IV
IMPLÉMENTATION ET RÉSULTATS

Après avoir spécifié notre protocole de sécurité par un clone de référence, nous avons effectué l'implémentation de ce dernier en restant le plus fidèle possible à sa spécification qui a été présentée au chapitre précédent. Pour ce faire, nous avons utilisé la plate-forme d'agents mobiles Grasshopper. En utilisant les fonctionnalités de cette dernière avec d'autres fonctionnalités de sécurité, nous avons implémenté une application du commerce électronique afin d'illustrer le fonctionnement de notre protocole. À l'issue de cette étape, nous avons testé la capacité de ce dernier à détecter les attaques contre l'agent mobile et évalué sa performance en terme de temps d'exécution et de trafic généré. Dans ce chapitre, nous allons tout d'abord présenter l'environnement de l'implémentation de notre protocole, puis la façon dont l'implémentation de ce dernier a été effectuée. Par la suite, nous allons décrire les tests réalisés, détailler les mesures effectuées et les analyser pour finalement déduire la performance de notre protocole.

4.1 Environnement de l'implémentation

Dans le but de réaliser un prototype de notre protocole, nous avons eu à choisir un environnement approprié permettant d'illustrer les différentes fonctionnalités de ce dernier. Pour ce faire, nous avons étudié les fonctionnalités de la plate-forme Grasshopper utilisée dans notre laboratoire et analysé sa capacité à fournir les outils nécessaires à l'implémentation de notre protocole. Parmi les exigences requises, nous pouvons citer : la capacité de supporter une application du commerce électronique, la possibilité de réaliser des communications locales et distantes, et la capacité de supporter plusieurs protocoles pour une meilleure interopérabilité.

4.1.1 Description de l'application utilisée

Afin d'illustrer le fonctionnement de notre protocole dans un contexte proche de la réalité, nous avons choisi une application de recherche du meilleur prix d'un certain produit. Pour ce faire, l'utilisateur configure l'agent mobile en lui donnant une description de l'article recherché, une stratégie de négociation contenant entre autres le prix maximum à accepter et un itinéraire de départ. Dans notre cas, l'itinéraire de départ comprend seulement la prochaine plate-forme à visiter. Une fois cette étape préliminaire franchie, l'agent mobile migre vers la première plate-forme de son itinéraire. Tel que décrit dans le chapitre précédent, chaque plate-forme est représentée auprès des entités extérieures par un agent stationnaire. Ainsi, quand l'agent mobile *AM* arrive dans une plate-forme P_{ij}, l'agent stationnaire AS_{ij} de cette dernière amorce son exécution. Il exécute alors la partie du code concernant la négociation puis celle du choix de la prochaine migration. Finalement, il fait migrer *AM* vers la prochaine plate-forme de son itinéraire précédemment choisie. Une fois qu'un certain nombre de plates-formes visitées est atteint ou que le prix trouvé est convenable, *AM* revient à sa plate-forme d'origine.

4.1.2 La plate-forme Grasshopper

L'environnement de développement choisi, pour implémenter notre protocole, est la plate-forme d'agents mobiles Grasshopper [GRA01]. Cette plate-forme a été développée par la société allemande IKV, dont la première version était disponible en août 1998. Son utilisation est gratuite pour des fins de recherche. Le langage de développement de cette dernière est le langage Java, principalement en raison de sa portabilité. La programmation des agents mobiles se fait donc aussi en Java. Grasshopper est conforme au standard de l'OMG (Object Management Group) sur les agents mobiles, qui est MASIF (Mobile Agent System Interoperability Facility). Ce dernier a été conçu afin d'assurer l'interopérabilité entre des plates-formes d'agents mobiles de différents vendeurs.

Grasshopper est un environnement d'agents répartis. Il est composé d'agents, de places, d'agences et de régions. Les agents sont de deux types; ils peuvent être mobiles ou statiques. Les agents mobiles ont la capacité de se déplacer d'une place à une autre afin d'effectuer certaines tâches. Les agents statiques ou stationnaires n'ont pas cette capacité et ils sont condamnés à résider dans la même place tout au long de leur existence. L'agence représente l'environnement d'exécution des agents mobiles et stationnaires, et peut contenir plusieurs places. Au moins une agence doit s'exécuter sur un hôte afin de pouvoir exécuter un agent. Une place constitue un regroupement logique de certaines fonctionnalités à l'intérieur d'une même agence. Quant à la région, elle est utilisée pour avoir une vision globale des entités réparties précédemment citées, facilitant ainsi leur gestion dans l'environnement Grasshopper. Les agences avec leurs places peuvent être associées à une région spécifique, en s'enregistrant auprès de cette dernière lors de leur création. La région enregistre par la suite automatiquement chaque agent s'exécutant dans une agence précédemment enregistrée. Si un agent migre vers une autre agence, les informations d'enregistrement sont automatiquement mises à jour dans la région. La Figure 4.1 illustre la structure hiérarchique des composantes de Grasshopper.

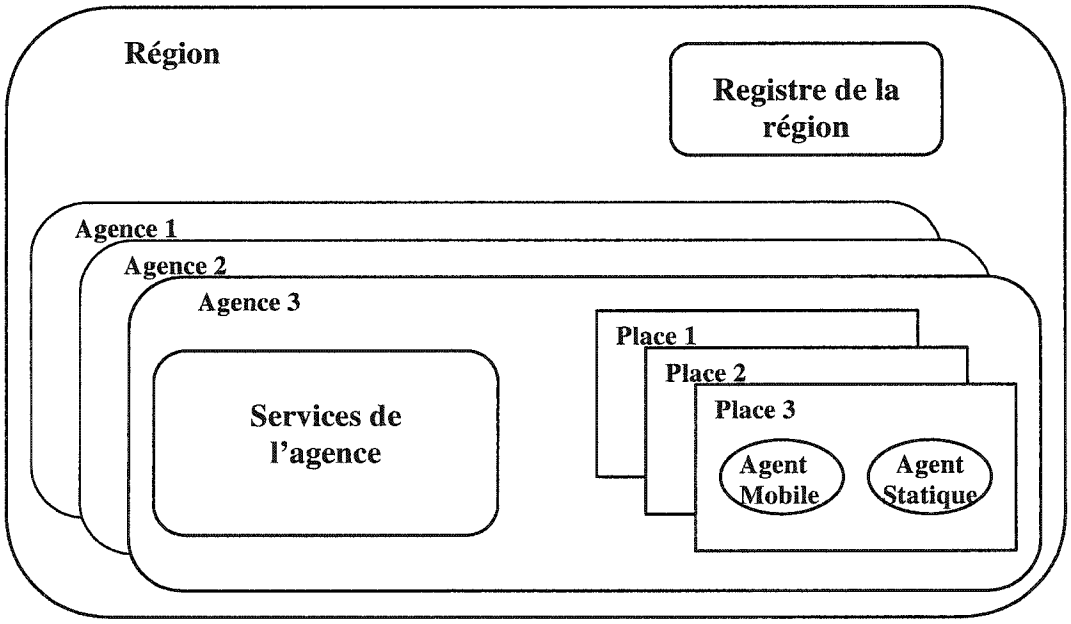


Figure 4.1 Structure hiérarchique des composants de Grasshopper

4.1.3 La communication dans Grasshopper

Grâce à son service de communication, Grasshopper intègre la migration de l'agent et la communication qui se font avec des appels de procédures à distance à travers le réseau. La communication entre deux entités se fait de manière transparente, i.e. sans que l'une connaisse la localisation de l'autre, quand une région Grasshopper est utilisée et que les agences s'exécutant sont enregistrées auprès cette dernière. Plus précisément, la communication entre deux agents *1* et *2* se fait selon les étapes indiquées à la Figure 4.2. Tout d'abord, un objet dit « *proxy* » est créé par *agent 1* afin de contacter *agent 2;* cet objet est une instance de l'interface de *agent 2* (Étape 1). La région est alors contactée par ce *proxy* afin de localiser *agent 2* (Étape 2), puis *agent 1* communique avec ce dernier en invoquant une de ses méthodes à travers le service de communication (Étape 3). L'invocation est ensuite acheminée par un protocole supporté par les deux agences sur le canal de communication (Étape 4). Finalement la méthode en question est invoquée localement dans *agence 2* (Étape 5).

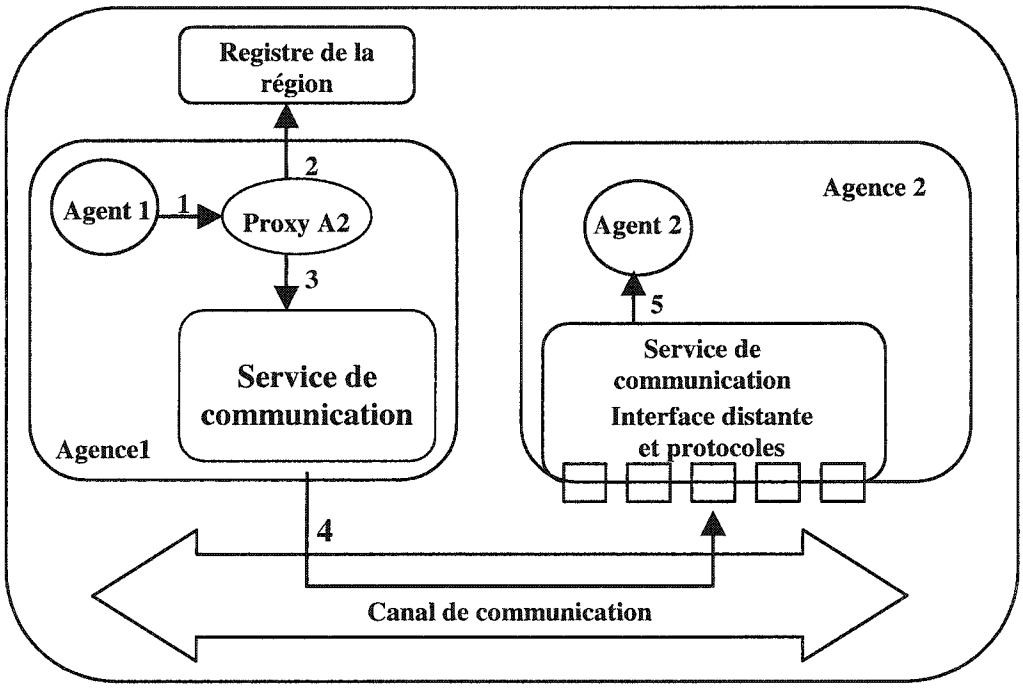


Figure 4.2 Communication entre deux agents dans Grasshopper

Grasshopper offre deux principaux modes de communication, synchrone et asynchrone. Dans la communication synchrone qui est aussi appelée communication bloquante, quand un agent invoque une méthode d'un agent distant, il bloque jusqu'à ce que cette dernière retourne le résultat. Par contre, dans la communication asynchrone, l'agent peut continuer son exécution et vérifier ultérieurement le résultat retourné. Cette communication est appelée aussi communication non-bloquante. Dans ce cas, le résultat retourné peut être récupéré de trois façons différentes. Avec la première façon, l'agent peut effectuer sa tâche en parallèle et, quand il a besoin du résultat, il consulte si la méthode a retourné ce dernier. Selon la deuxième, l'agent vérifie périodiquement si la méthode a retourné un résultat. Et selon la dernière, l'agent demande à être notifié quand le résultat arrive. Selon les exigences de l'application, l'un de ces types de communication pourra être utilisé.

4.1.4 Les protocoles supportés par Grasshopper

La communication constitue un concept important de notre implémentation et y est utilisée à plusieurs reprises. Grasshopper permet d'établir des communications transparentes entre deux agents, distants ou non, grâce à son service de communication. Ce service propose les protocoles de communications suivants :

- CORBA IIOP (Common Object Request Broker Architecture – Internet Inter ORB Protocol) : Ce protocole peut être utilisé dans tous les environnements supportant CORBA, indépendamment de l'implémentation ORB du fournisseur. Il utilise un mécanisme conforme aux standards pour se connecter à un objet utilisant un service de nom CORBA.
- MAF IIOP : Ce protocole est une spécialisation du protocole CORBA IIOP développé pour l'interaction entre des systèmes d'agents. Il a été décrit dans le standard MASIF (Mobile Agent System Interoperability Facility) et assure la connectivité entre des systèmes d'agents mobiles de différents fournisseurs. Ainsi, MAF IIOP n'utilise pas le service de communication de Grasshopper et se connecte directement à l'interface MASIF de l'entité avec laquelle il communique.

- RMI (Remote Method Invocation) : L'invocation de méthode à distance introduite avec le JDK 1.1 permet à des objets Java d'invoquer des méthodes d'autres objets Java s'exécutant dans une autre machine virtuelle. Ce protocole est inclus dans chaque machine virtuelle compatible avec JDK 1.1 et est utilisé par défaut par les agences Grasshopper.
- RMI avec SSL : En utilisant ce protocole, RMI s'exécute au-dessus de sockets protégées par le protocole SSL (Secure Socket Layer), ce qui permet de protéger toutes les données. Pour utiliser ce protocole, un package de sécurité supplémentaire est nécessaire.
- Socket : La communication par socket est la façon la plus rapide d'effectuer une interaction distante. Cette technique est robuste et évite le surcoût du modèle d'objets répartis. Ce protocole est utilisé par défaut par les agences Grasshopper.
- Socket sur SSL : En utilisant ce protocole, les sockets sont protégés par SSL. Les conditions de son utilisation sont les mêmes que celles citées pour RMI/SSL.

Le protocole SSL (Secure Socket Protocol) [FRE96] est un standard utilisé dans Internet pour effectuer des communications sécuritaires. Il permet d'assurer l'authentification, l'intégrité des données ainsi que leur confidentialité. Lors de l'ouverture de la session « *handshake* », les deux entités communicantes s'authentifient en s'échangeant leurs identités et se mettent d'accord sur le mode d'encryptage et sur l'algorithme à utiliser. Par la suite, les données à envoyer sont fragmentées en des blocs faciles à manipuler et éventuellement compressées, puis un *MAC* (Message Authentification Code) leur est appliqué. Elles sont ensuite encryptées pour finalement être envoyées. Les données reçues subissent les opérations inverses, i.e. elles sont décryptées, vérifiées, décompressées, assemblées puis délivrées aux couches supérieures. Ainsi, l'intégrité est assurée par l'utilisation du code d'authentification des messages et la confidentialité par l'encryptage. Cette dernière peut se faire soit avec un algorithme d'encryptage symétrique ou asymétrique qui aura été négocié lors du *handshake*. Les identités des entités sont généralement échangées par l'intermédiaire

d'un certificat numérique qui suit la norme X.509 v3. Ces certificats sont acceptés s'ils sont signés par une autorité de certification commune.

Dans notre implémentation, nous avons opté pour l'utilisation de SSL afin d'assurer la sécurité des communications et le transfert des agents. En effet, il est important que les communications entre les différents agents soient sécuritaires, étant donné que des informations sensibles sont envoyées à travers le réseau. Ces informations concernent les données utilisées par le clone pour vérifier la bonne exécution de l'agent mobile. Elles sont de ce fait essentielles pour le bon fonctionnement de notre protocole et on doit s'assurer de leur intégrité, de leur confidentialité et aussi de leur non répudiation. D'où la nécessité d'utiliser SSL. De même, le transfert sécuritaire des agents mobiles est indispensable pour avoir une sécurité globale. En effet, étant donné que notre protocole se concentre sur la détection des attaques de la part des plates-formes malicieuses, SSL le complémente puisqu'il prévient les attaques intermédiaires qui pourraient survenir lors du transport des agents.

4.2 Implémentation du protocole

Après avoir étudié l'environnement Grasshopper et ses fonctionnalités, nous avons procédé à la réalisation de notre protocole, en restant les plus fidèles possibles à sa spécification. Nous avons tout d'abord cherché à réaliser les hypothèses émises qui sont indispensable à la réalisation de ce dernier, puis nous avons implémenté les différentes entités participantes ainsi que leurs fonctionnalités. Une attention particulière a été portée à la communication, à l'encryptage et à la signature numérique étant donné le rôle joué par ces dernières dans notre protocole. L'implémentation a été réalisée en Java en raison de sa portabilité et sa compatibilité avec la plate-forme Grasshopper.

4.2.1 Mise en œuvre des hypothèses

Notre première hypothèse était que le réseau est décomposé en régions et que, dans chaque région, il y a un serveur de confiance. La mise en œuvre de cette hypothèse a été réalisée en se basant sur des notations pertinentes et en y tenant compte dans

l'implémentation. Ainsi, on a utilisé les indices de l'agence pour savoir à quelle région elle appartenait. Le premier indice indique le numéro de l'agence et le deuxième la région à laquelle cette dernière appartient. Par exemple, $Agence_{ij}$ indique que c'est la $i^{ème}$ agence de l'itinéraire de l'agent mobile et qu'elle appartient à la région *j*. Cette dernière sait donc qu'elle fait partie de la région *j* du réseau et qu'elle fait affaire par conséquent avec le serveur de confiance SC_j. Il ne faut pas confondre la notion de région de notre protocole avec le concept de région de Grasshopper. Dans notre protocole, une région correspond à une décomposition logique du réseau, et le serveur de confiance est une agence comme les autres, qui appartient à cette région, mais qui a un rôle différent. La région dans Grasshopper est par contre une entité au même titre que les agences et qui a un rôle à jouer, comme par exemple, permettre la communication entre des agents d'une même région. Dans notre protocole, nous avons choisi d'utiliser une seule région dans laquelle sont enregistrées toutes les agences utilisées pour faciliter les communications. L'affectation d'une région Grasshopper pour chaque région du réseau est possible mais aurait compliqué l'implémentation sans pour autant lui ajouter de la valeur.

Dans la deuxième hypothèse, nous avons supposé que chaque entité participante possédait une clé publique et une clé privée. Cette hypothèse a été réalisée en générant ces clés au moyen de l'outil *keytool*. Cet outil est disponible chez Sun et permet de générer des clés afin d'utiliser l'encryptage et la signature numérique. Ainsi, nous avons généré une paire de clés publique/privé pour chaque agence intervenant dans notre protocole. Étant donné que les agences sont représentées par leurs agents stationnaires, ce sont ces derniers qui détiennent ces clés et qui les manipulent. La taille des clés générées est de 1024 bits, l'algorithme d'encryptage choisi est *RSA* et celui de signature numérique est *MD5withDSA*. Ces choix sont considérés comme sécuritaires au moment de la rédaction du mémoire. Les clés générées sont stockées dans le fichier *keystore* qui est localisé par défaut dans le compte de l'utilisateur et est protégé par un mot de passe choisi lors de sa création. Un mot de passe supplémentaire doit être spécifié lors de la création de chaque clé privée et est indispensable pour utiliser cette dernière par la suite.

Afin de faciliter les échanges des clés publiques, nous les avons toutes mises dans chaque fichier *keystore* de chaque station utilisée.

Quant à la réalisation de la troisième hypothèse, nous avons utilisé SSL qui est disponible dans Grasshopper pour sécuriser les communications entre les différents agents ainsi que leur transport. Dans Grasshopper, SSL est utilisé conjointement avec les certificats X.509 afin d'assurer la confidentialité et l'intégrité des données ainsi que l'authentification mutuelle. Le fonctionnement de SSL a déjà été détaillé auparavant.

4.2.2 Implémentation des agents AS_{01}, AM, AS_{ij} et AS_j

Notre protocole implique l'intervention de plusieurs entités, comme il a été présenté au chapitre précédent. Les plates-formes notées P_{ij} dans ce dernier ainsi que les serveurs de confiance sont représentés avec Grasshopper par des agences. Les agences représentant les plates-formes des vendeurs sont alors notées $Agence_{ij}$ et celles représentant les serveurs de confiance continuent à être notées SC_j. Nous allons, dans ce qui suit, détailler l'implémentation de l'agent stationnaire AS_{01} représentant la plate-forme d'origine $Agence_{01}$, celle de l'agent mobile *AM*, l'implémentation des agents stationnaires AS_{ij} représentant les agences $Agence_{ij}$ ainsi que l'implémentation de AS_j représentant les serveurs de confiances SC_j.

L'agent stationnaire AS_{01}

L'agent stationnaire AS_{01} est implémenté par la classe *AgentStatique01* qui hérite de la classe *StationnaryAgent* définie par Grasshopper (c.f. Figure 4.3). Cet héritage lui permet d'utiliser les fonctionnalités de cette classe qui, elle-même, hérite de la classe générique *Agent*. La classe *AgentStatique01* implémente l'interface *IListeningAgentS01* qui lui permet de détecter les évènements se produisant dans l'$Agence_{01}$ comme l'ajout ou la suppression d'un agent et ce, grâce à la méthode *eventA01Detected()*. La méthode *init()* de AS_{01} a été redéfinie afin d'initialiser les variables utilisées par cette classe. Plusieurs méthodes ont été créées pour réaliser ses différentes fonctionnalités et ont été utilisées dans la méthode *live()*. Ces méthodes sont scindées en deux groupes.

Le premier groupe concerne les méthodes utilisées avant la migration de l'agent mobile et de son clone, alors que le deuxième concerne celles utilisées après que ces derniers soient revenus. Dans le premier groupe de méthodes, on peut distinguer celles de la création de l'agent et de son clone ainsi que celles de la création de leurs proxies afin de pouvoir communiquer avec eux, comme expliqué auparavant. Afin de distinguer *AM* et son clone, nous avons utilisé la notion de propriété définie dans Grasshopper en attribuant des propriétés différentes a chacun d'eux. La création des proxies a été faite en utilisant des objets de l'interface *IAgentMobile* afin de pouvoir accéder aux méthodes de *AM* et de son clone. De plus, un objet de la classe *ListenerGHA01* a été utilisé afin d'utiliser ses méthodes permettant de détecter l'ajout et la suppression des agents. Celles-ci ont été d'une grande utilité notamment pour mesurer le temps d'exécution du protocole.

Le deuxième groupe de méthodes intervient principalement lorsque *AM* et son clone reviennent de leurs migrations. Parmi ces méthodes, on peut citer celles qui servent à extraire les informations recueillies par *AM,* comme la liste des agences visitées et les prix recueillis dans chacune d'entre elles. D'autres méthodes de ce groupe incluent la vérification des signatures et le décryptage des prix recueillis, ainsi qu'une méthode servant à récupérer la liste des agences coupables éventuelles. L'utilisation de ces méthodes fait appel encore une fois à l'interface de l'agent mobile *IAgentMobile*. Par ailleurs, l'encryptage et la signature numérique utilisent des objets des classes *KeyStoreHandler* et *Message*. La façon dont l'encryptage et la signature numérique ont été implémentés est explicitée par la suite. La Figure 4.3 présente le diagramme des classes de *AgentStatique01*.

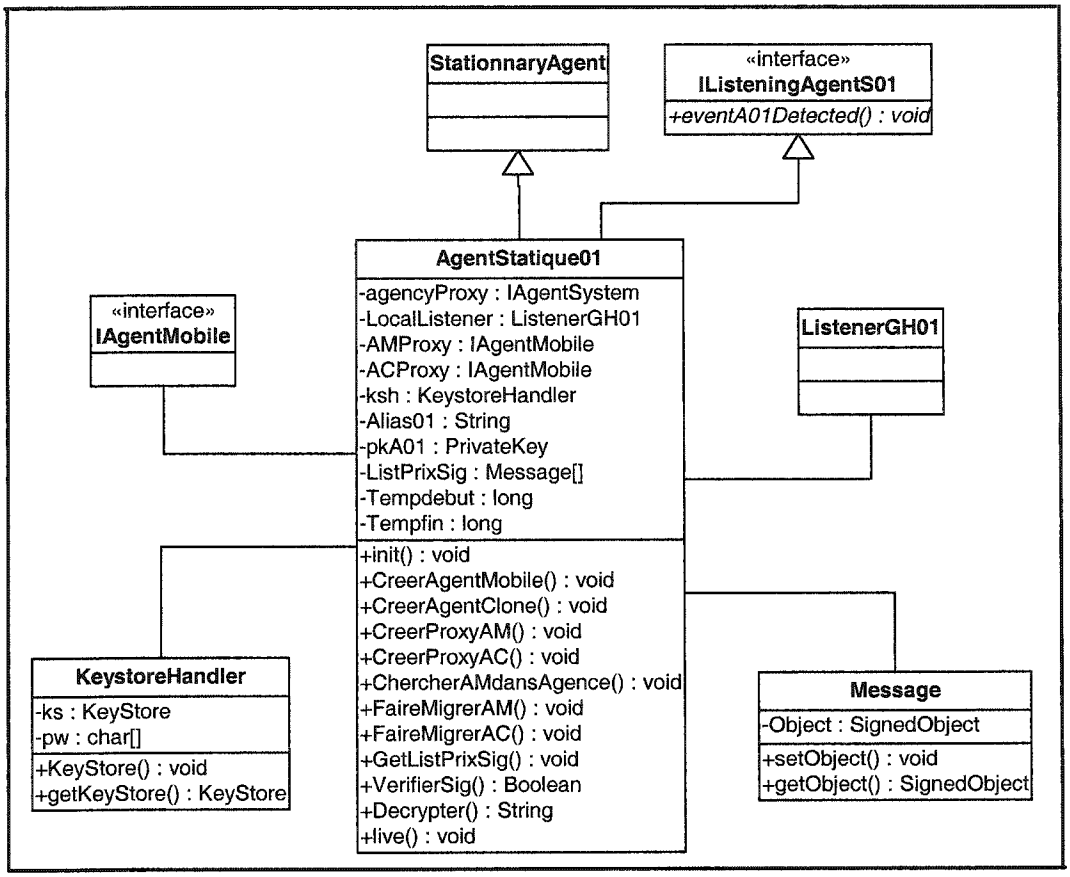


Figure 4.3 Diagramme des classes de l'agent *AgentStatique01*

L'agent mobile AM

L'agent mobile *AM* est implémenté par la classe *AgentMobile* qui hérite de la classe *MobileAgent* qui, elle-même, hérite de la classe *Agent*. Cet héritage lui permet d'utiliser les fonctionnalités de Grasshopper offertes à ces classes, comme la migration par exemple. La classe *AgentMobile* implémente l'interface *IAgentMobile* dans laquelle sont définies les méthodes pouvant être accédées par d'autres agents. La méthode *init()* de *AgentMobile* initialise les différentes variables utilisées par cette dernière, comme celles utilisées pour stocker les prix ou les agences visitées. Cette méthode est exécutée une seule fois, lors de la création de *AM*.

Plusieurs méthodes ont été créées pour permettre à l'agent d'effectuer sa tâche. La plupart de ces méthodes font partie de son interface, étant donné que ce sont les agents

stationnaires de chaque agence qui les invoquent quand *AM* leur demande. Le choix d'exécuter *AM* par invocation s'est imposé par le fait que son clone est une copie exacte de ce dernier mais ne doit pas avoir exactement le même comportement que celui-ci. En effet, le clone n'est pas autonome dans son exécution et attend que le serveur de confiance l'exécute, quand ce dernier reçoit les informations nécessaires. De plus, *AM* et son clone n'ont pas le même itinéraire et chacun d'entre eux le définit dynamiquement. De ce fait, si on ne contrôle pas l'exécution du clone, ce dernier va suivre *AM* et faire la même chose que lui puisqu'il est sa copie exacte. L'utilisation de l'invocation permet de régler ce problème mais contribue à augmenter sensiblement le trafic, étant donné qu'à chaque invocation la région est contactée afin de localiser l'agent invoqué.

Parmi les méthodes de *AM*, nous pouvons citer la méthode *Negociation()* qui simule sa stratégie de négociation. Cette méthode est invoquée par l'agent vendeur avec lequel *AM* négocie jusqu'à ce qu'un compromis de prix soit conclu entre les deux parties. À titre d'illustration, la stratégie de négociation utilisée est de proposer un prix initial raisonnable et de l'augmenter de 2% à chaque étape. Une autre méthode concerne le rajout du prix convenu entre les deux parties à la liste des prix proposés, une fois que ce dernier a été encrypté et signé par le vendeur. Cette méthode a nécessité l'utilisation de la classe *Message* (c.f. Figure 4.4). Une autre méthode importante dans notre protocole est celle qui permet de faire le choix de la prochaine migration. Compte tenu du temps limité dont nous disposions, celle-ci a été simplifiée et les choix de migration sont prédéfinis de façon statique, contrairement à ce que nous avons indiqué dans le chapitre précédent. Dans la méthode *MigrerAgent()* qui sert à faire migrer l'agent mobile, il a fallu faire attention de ne pas faire migrer l'agent avant qu'il finisse sa tâche; pour ce faire, nous avons utilisé une mise en veille et une synchronisation des « *threads* ». En effet, quand l'agent reçoit une demande de migration, un nouveau « *thread* » est créé pour exécuter cette méthode pendant que le « *thread* » principal continue d'exécuter l'agent. À ce moment là, nous avons mis en veille le « *thread* » demandant la migration jusqu'à ce que le « *thread* » principal finisse sa tâche. À ce

moment là, ce dernier notifie le premier « *thread* » et la migration se fait au bon moment. La Figure 4.4 présente le diagramme des classes de *Agent Mobile.*

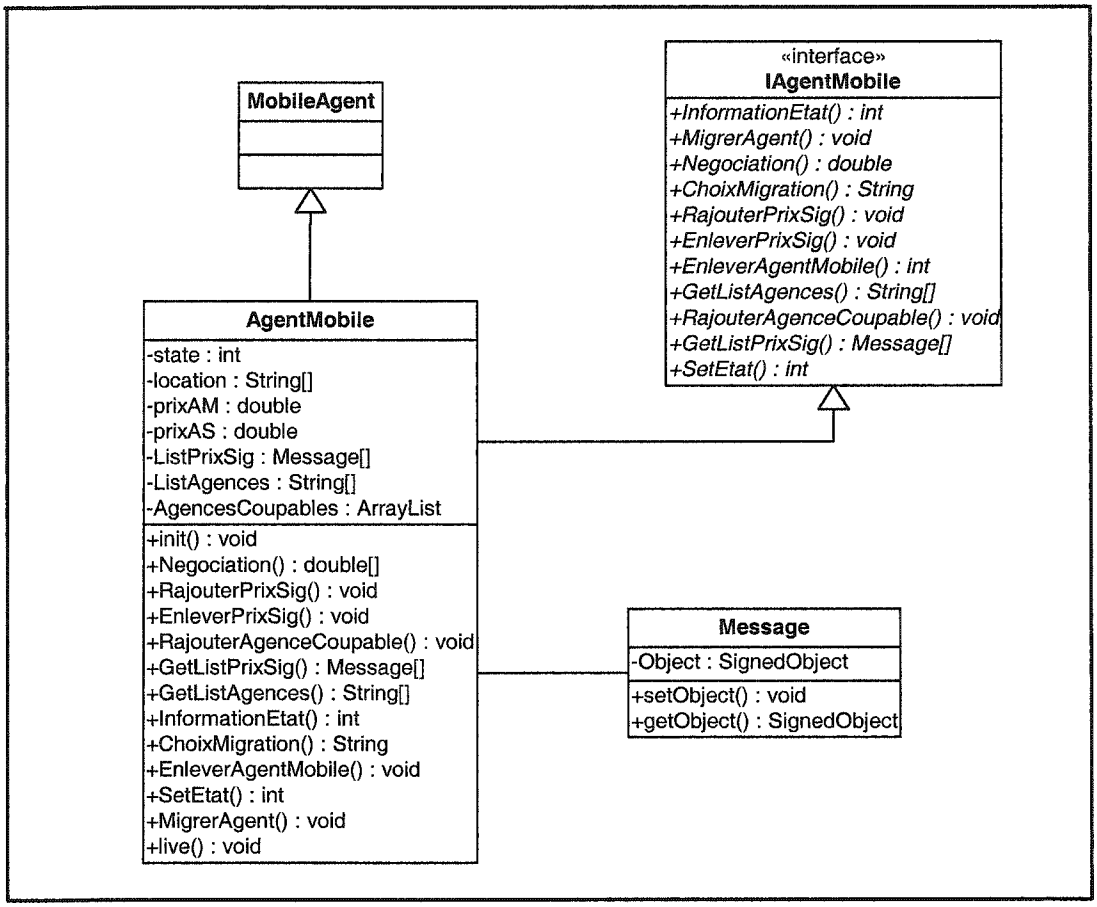


Figure 4.4 Diagramme des classes de l'agent *AM*

L'agent stationnaire AS_{ij}

L'agent stationnaire AS_{ij} représentant l'agence $Agence_{ij}$ a été implémenté par la classe *AgentStatiqueij* qui hérite de la classe *StationnaryAgent*, qui elle-même hérite de la classe générique *Agent*. Cet héritage lui permet de bénéficier de plusieurs fonctionnalités des agents définies dans Grasshopper. La classe *AgentStatiqueij* implémente l'interface *ILesteningAgentSij* lui permettant de détecter les évènements se produisant dans $Agence_{ij}$ comme l'ajout ou la suppression d'un agent et ce, grâce à la méthode *eventAijDetected()*. La méthode *init()* de AS_{ij} a été redéfinie afin d'initialiser les

variables utilisées par cette classe, comme par exemple le prix des produits, ou encore, les clés privée et publique utilisées par cette dernière.

Dans cette classe, plusieurs méthodes ont été créées afin de simuler le comportement du vendeur. La méthode *ChercherAMdansAgence()* permet de chercher l'agent mobile *AM* dans $Agence_{ij}$ une fois que son arrivée ait été détectée. Cette détection se fait grâce à l'objet *localListener* de la classe *ListenerGHAij*. Cette classe est expliquée plus en détail par la suite. La recherche de *AM* dans $Agence_{ij}$ se fait en appliquant un filtre par nom à la liste des agents s'exécutant dans cette agence. Grasshopper permet de créer un filtre par rapport à plusieurs caractéristiques de l'agent, comme son identificateur ou son nom. Quand l'agent est trouvé, la méthode *Negociation()* est exécutée. Cette dernière invoque la méthode *Negociation()* de *AM* en précisant le prix proposé pour le produit recherché par ce dernier. L'invocation se fait grâce à l'interface *IAgentMobile*. La négociation se fait jusqu'à ce qu'un compromis soit trouvé. Le prix convenu est ensuite encrypté avec la clé publique de l'agence d'origine $Agence_{01}$ puis signé numériquement avec la clé privée de AS_{ij}. La façon dont l'encryptage et la signature numérique se font est expliquée plus en détail par la suite. Le choix de la migration est ensuite fait en invoquant la méthode *ChoixMigration()* de *AM*. Par la suite, une méthode permet de chercher l'agent stationnaire du serveur de confiance de la région grâce à un filtre Grasshopper, appliqué cette fois-ci sur tous les agents de la région. Ceci permet de lui communiquer les informations relatives à l'exécution de *AM*. La façon dont la communication se fait est expliquée plus en détail par la suite. Les informations à envoyer sont extraites de *AM* grâce à des invocations sur ce dernier. Finalement, une dernière invocation est faite afin de le faire migrer à la destination choisie auparavant. Le diagramme des classes de *AgentStatiqueij* est illustré à la Figure 4.5.

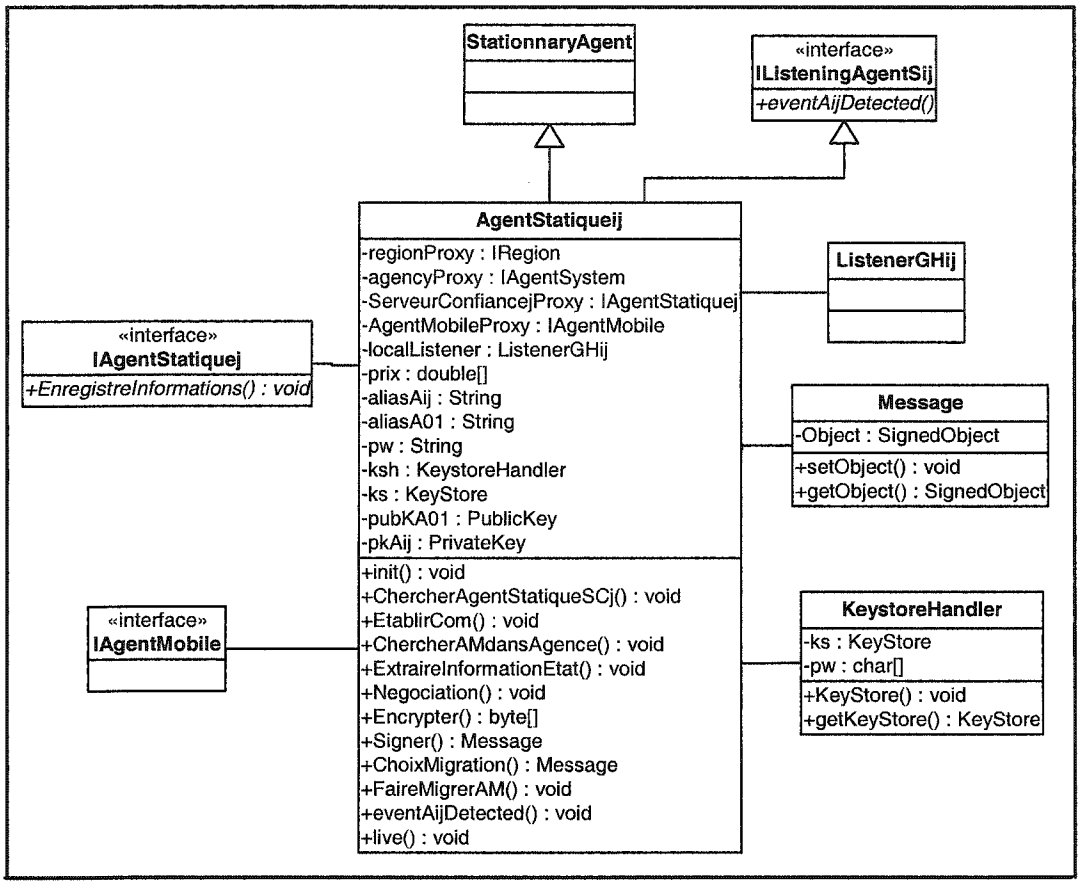


Figure 4.5 Diagramme des classes de l'agent *AgentStatiqueij*

L'agent stationnaire AS_j

L'agent stationnaire AS_j représentant le serveur de confiance SC_j a été implémenté par la classe *AgentStatiquej* qui hérite de la classe *StationnaryAgent*. Cet héritage lui permet de bénéficier de plusieurs fonctionnalités des agents définies dans Grasshopper. La classe *AgentStatiquej* implémente deux interfaces : l'interface *ILesteningAgentSij* permettant à AS_j de détecter les évènements se produisant dans l'$Agence_{ij}$ comme l'ajout ou la suppression d'un agent, et l'interface *IAgentStaitquej* à travers laquelle d'autres agents peuvent communiquer avec lui. La méthode *init()* de AS_j a été redéfinie pour initialiser les variables utilisées par cette classe. Plusieurs méthodes ont été créées pour réaliser ses différentes fonctionnalités et ont été utilisées dans la méthode *live()*. Cette exécution est scindée en deux parties logiques.

La première partie comprend les méthodes servant à simuler le comportement du vendeur en exécutant le clone de *AM*. La méthode *EnregistreInformations()* enregistre les informations d'exécution de *AM* au fur et à mesure qu'elles arrivent. Par ailleurs, la méthode *ChercherACdansAgence()* permet de chercher le clone de *AM* dans le serveur de confiance SC_j, une fois que l'arrivée de ce dernier ait été détectée. Cette détection se fait grâce à l'objet *localListener* de la classe *ListenerGHAij*. Cette classe est expliquée plus en détail par la suite. La recherche du clone se fait en appliquant un filtre comme expliqué précédemment. Par la suite, l'exécution de *AM* est vérifiée en exécutant *AC* avec les informations reçues. La méthode *Negociation()* est alors exécutée, qui elle-même invoque la méthode *Negociation()* du clone en utilisant l'interface *IAgentMobile*. Cette interface est commune à *AM* et à son clone puisque ce dernier est une copie exacte de *AM*. Le résultat de la négociation est ensuite protégé par encryptage et authentifié par signature numérique puis rajouté à la liste des prix du clone. Ceci est fait dans le but de prévenir le cas d'attaque où on aurait besoin d'envoyer une copie du clone à la place de *AM*. Par la suite, le choix de la migration est aussi simulé avec la méthode *ChoixMigration()* en invoquant le clone par l'intermédiaire de son interface. Finalement, les informations d'état son extraites du clone par invocation, avec la méthode *ExtraireInformationEtat()*, puis comparées à celles précédemment reçues résultant de l'exécution de *AM*.

Dans la deuxième partie de l'exécution de AS_j, les différentes vérifications sont effectuées afin de détecter un éventuel comportement malicieux. Ainsi, tout d'abord un premier test est effectué pour savoir si *AM* a été envoyé au bon endroit. Ceci a été fait en comparant l'identité de la plate-forme courante avec celle trouvée par le clone dans l'exécution précédente. Un deuxième test est ensuite effectué s'il n'y a pas eu d'attaque détectée lors du test précédent. Dans ce test, nous vérifions que la signature sur l'état d'entrée n'a pas été forgée. Finalement, s'il n'y a pas eu encore d'attaque détectée, nous vérifions que les états d'entrée et de sortie trouvés par le clone sont bien égaux à ceux trouvés par *AM*. Si une attaque est détectée suite à ces tests, l'agent mobile *AM* est

localisé, puis la méthode *AvorterExecutionAgent()* est exécutée selon l'agence qui a été désignée coupable.

Dans cette méthode, AS_j essaie tout d'abord d'avorter l'exécution de *AM* en invoquant la méthode *EnleverAgentMobile()* et ce, à cinq reprises. Il est important d'essayer de faire cette invocation plusieurs fois car *AM* peut être en pleine migration pendant qu'il est invoqué, auquel cas l'invocation ne va pas fonctionner. En l'invoquant par contre plusieurs fois et en faisant un temps d'attente d'une seconde entre chaque invocation, elle finit par fonctionner. Par la suite, les informations reçues à partir du moment où *AM* a été corrompu sont enlevées de l'espace de stockage, l'agence coupable est rajoutée à la liste des agences coupables et le clone est remis à l'état auquel il était après son exécution dans la dernière plate-forme non-malicieuse. Ceci a été réalisé en invoquant respectivement la méthode *remove()* de chaque liste de stockage et les méthodes *RajouterAgenceCoupable()* et *SetEtat()* du clone à partir de son interface. Finalement, un nouvel agent mobile est créé en faisant une copie du clone et est migré vers l'agence suivant l'agence coupable dans l'itinéraire de l'agent initial. La Figure 4.6 présente le diagramme des classes de *AgentStatiquej*.

Les classes $ListenerGHA_{ij}$

Les classes des agents précédemment citées ont utilisé la classe $ListenerGHA_{ij}$ (respectivement $ListenerGHA_{0l}$ et $ListenerGHSC_j$) afin de détecter l'arrivée ou le départ de l'agent mobile et/ou de son clone. Cette classe implémente l'interface *ISystemListener* qui possède des méthodes prédéfinies. Cette interface permet à notre classe $ListenerGHA_{ij}$ (respectivement $ListenerGHA_{0l}$ et $ListenerGHSC_j$) de redéfinir ses méthodes afin de les adapter à notre protocole. Les méthodes de cette interface sont: *agencyAdded(), agencyRemoved(), agentAdded(), agentChanged(), agentRemoved(), placeAdded(), placeChanged()* et *placeRemoved()*. Ces méthodes servent respectivement à notifier : l'ajout ou la suppression d'une agence; l'ajout, la modification ou la suppression d'un agent; l'ajout, la modification ou la suppression d'une place (c.f. Figure 4.7). La notification est faite à l'agent qui a implémenté l'interface

IlisteningAgentSij (respectivement *IlisteningAgentS01* et *IListeningAgentSj*) du fait de l'utilisation de l'objet *localListener* de la classe *ListenerGHA$_{ij}$* (c.f. Figures 4.3, 4.5 et 4.6).

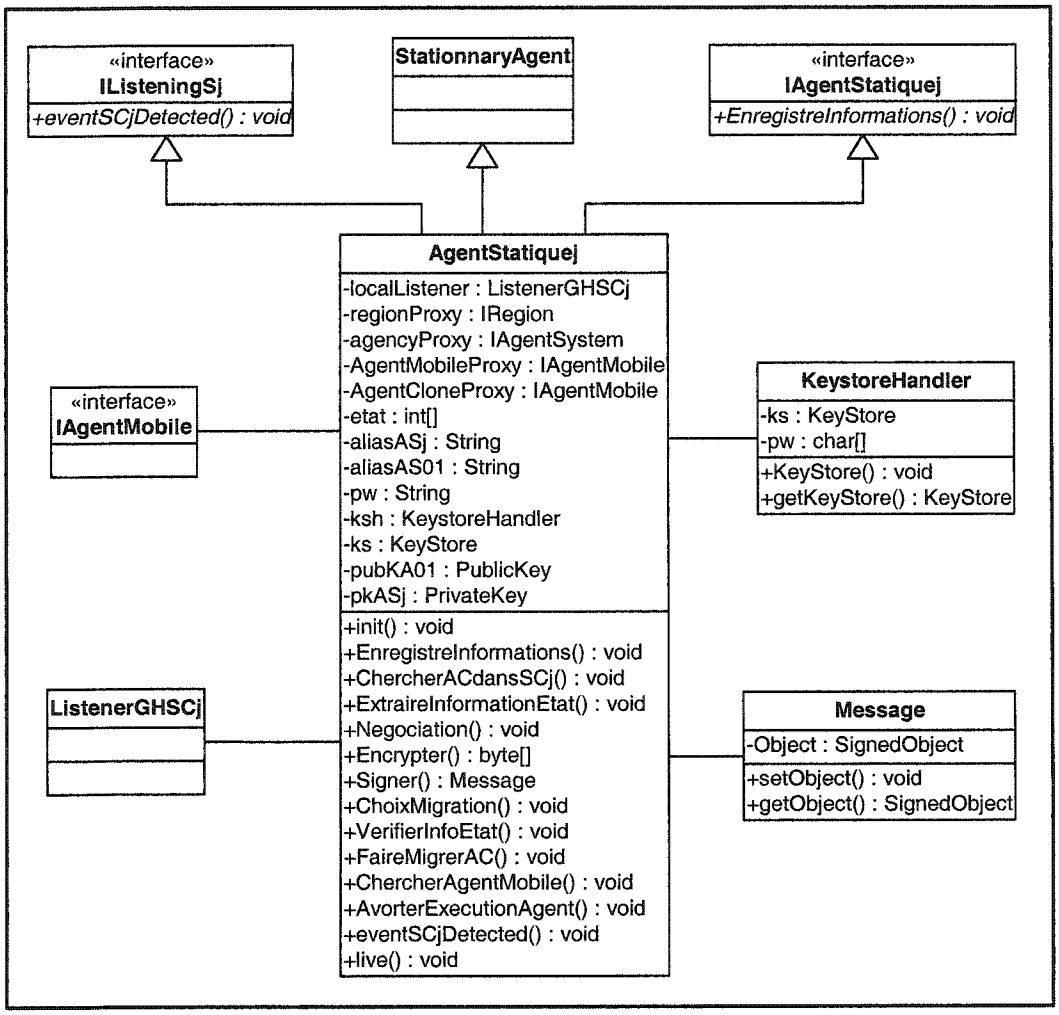


Figure 4.6 Diagramme des classes de l'agent *AgentStatiquej*

Cette interface contient la méthode *eventAijDetected()* (respectivement *eventA01Detected()* et *eventSCjDetected()*) qui est notifiée lorsqu'un des évènements précédemment cités se produit. Dans notre protocole, nous nous sommes intéressés seulement à l'ajout et à la suppression d'agent.

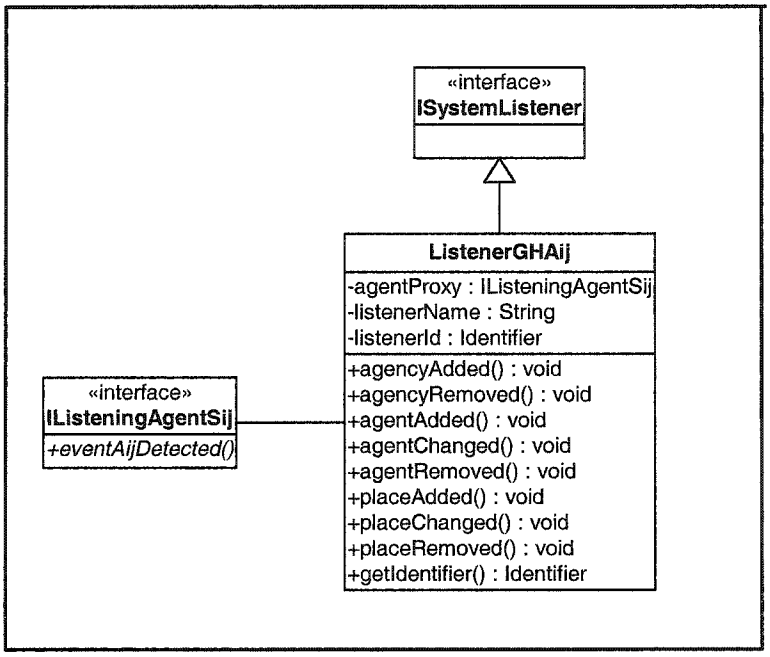


Figure 4.7 Diagramme des classes de *ListenerGHA$_{ij}$*

4.2.3 Implémentation de la communication

Dans notre implémentation, nous avons utilisé la communication à deux niveaux. Tout d'abord, elle a été utilisée par les agents stationnaires AS_{ij} pour exécuter l'agent mobile dans chaque agence et par les agents stationnaires AS_j pour exécuter le clone dans les serveurs de confiance. Dans un deuxième niveau, la communication a été utilisée pour que AS_{ij} communique les informations qui ont servi à exécuter *AM* à AS_j pour que ce dernier exécute le clone. Dans les deux cas, la communication se fait par invocation, comme expliqué auparavant, mais en utilisant des interfaces différentes. D'autres communications ont été utilisées quand il a fallu chercher un agent en particulier dans une agence ou dans la région Grasshopper.

Afin de pouvoir être exécuté, *AM* a implémenté une interface dans sa définition de classe (c.f. Figure 4.4). Dans cette interface, différentes méthodes ont été incluses comme celles servant à la négociation, le choix de la migration, l'extraction de l'état, le rajout et le retrait d'un prix. Afin de pouvoir invoquer ces méthodes, les agents stationnaires AS_{ij} et AS_j ont utilisé un objet « *proxy* », présenté dans la section 4.1.3,

comme une instance de l'interface *IAgentMobile* (c.f. Figure 4.5 et Figure 4.6). De cette façon, toutes les méthodes de l'interface de *AM* peuvent être invoquées. Dans ces invocations, le mode synchrone a été utilisé, étant donné que l'exécution de AS_{ij} et de AS_j ne pouvait pas continuer tant que la méthode invoquée n'a pas retourné de résultats.

Pour ce qui est de la communication entre AS_{ij} et AS_j, la classe de AS_j a implémenté une interface contenant la méthode *EnregistreInformation()* permettant de récupérer les données d'exécution envoyées par AS_{ij} (c.f. Figure 4.6). Afin de pouvoir invoquer cette méthode, la classe de AS_{ij} a utilisé l'objet *ServeurConfianceProxy* comme une instance de l'interface *IAgentStatiquej*, comme présenté dans le diagramme des classes de AS_{ij} à la Figure 4.5. Cette invocation a été réalisée avec le mode asynchrone puisque la méthode ne retourne aucune valeur. De ce fait, AS_{ij} peut continuer son exécution pendant que les informations nécessaires sont communiquées à AS_j accélérant ainsi l'exécution.

Les autres communications utilisées sont celles relatives à la recherche d'un agent, soit dans une agence en particulier ou dans la région Grasshopper tout entière. Cette recherche est indispensable pour créer l'objet *proxy* sur l'agent avec lequel on veut communiquer. Cette recherche se fait en invoquant la méthode *listAgents()* de l'interface *IAgentSystem* dans le cas d'une agence, ou de l'interface *IRegion* dans le cas où on cherche l'agent dans toute la région Grasshopper. À la Figure 4.8 sont présentés les diagrammes des classes de ces deux interfaces qui sont prédéfinies dans Grasshopper. La méthode *listAgents()* renvoie tous les agents de l'agence ou de la région, selon le cas, et est gardé seulement celui qui correspond à un nom particulier, dans le cas de filtrage par nom que nous avons utilisé. Par la suite, l'identificateur de ce dernier pourra être récupéré et son interface utilisée pour créer un « *proxy* » et communiquer avec lui. Par ailleurs, l'interface *IAgentSystem* a été aussi utilisée pour créer tous les agents du protocole grâce à la méthode *createAgent()* ainsi que pour créer un clone d'un agent grâce à la méthode *copyAgent()*.

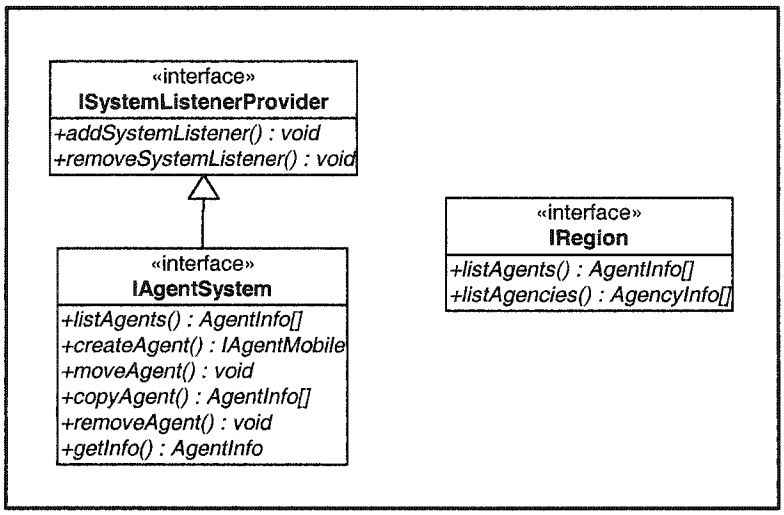


Figure 4.8 Diagrammes des classes des interfaces *IAgentSystem et IRegion*

4.2.4 Implémentation de l'encryptage et de la signature numérique

Afin d'assurer la confidentialité et la non répudiation des données résultant de la négociation, l'encryptage et la signature numérique ont été utilisées. Ainsi, avant de rajouter le prix convenu dans la liste des prix de l'agent *AM*, AS_{ij} l'encrypte d'abord avec la clé publique de l'agence d'origine *Agence01*, puis le signe numériquement avec sa clé privée.

Nous avons pour cela, utilisé les *API* de *JCE* (*Java Cryptography Extension*) [OAK01] fournies par Sun et celles développées par l'institut *IAIK* [IAI02]. Ces librairies permettent l'encryptage et le décryptage des données, la signature numérique et la vérification de cette signature, ainsi que la gestion des clés et des certificats. Comme présenté à la section 4.2.1, nous avons choisi l'algorithme *RSA* (Rivest, Shamir et Adelman) [PFL96] qui permet l'encryptage asymétrique. Pour la signature numérique, nous avons opté pour l'algorithme *MD5withRSA*. Ces deux algorithmes sont considérés par la communauté scientifique comme sécuritaires au moment de la rédaction de ce mémoire. Nous avons généré une paire de clé de 1024 bits pour chaque agent intervenant dans notre protocole, en utilisant l'utilitaire *keytool* de Sun. Toutes les paires de clés ainsi que leurs certificats sont sauvegardés dans un fichier appelé *keystore* et qui est localisé par défaut dans le répertoire de l'utilisateur.

Afin d'utiliser les fonctionnalités cryptographiques, la classe *KeyStoreHandler* a été utilisée (c.f Figure 4.5 et Figure 4.6). Cette dernière permet de récupérer les clés à utiliser en chargeant le fichier *keystore* et en fournissant les mots de passe appropriés. L'encryptage dans chaque agence et dans chaque serveur de confiance se fait par la fonction *Encrypter()*. Cette dernière, en utilisant les librairies de sécurité présentées précédemment et la clé publique de AS_{01}, prend en argument une chaîne de caractères représentant le prix à encrypter et renvoie un tableau d'octets comme résultat d'encryptage. Quant à la signature numérique, elle se fait par la méthode *Signer()* qui prend en argument le message à signer sous forme de chaîne de caractères et renvoie un objet de type *Message* que nous avons défini à partir de la classe *SignedObject*. La signature se fait ainsi en précisant l'algorithme de signature et en utilisant la clé privée appropriée précédemment chargée.

Pour récupérer les prix en clair et vérifier les signatures, AS_{01} utilise les opérations inverses implémentées avec les méthodes *VerifierSig()* et *Decrypter()*, tel qu'illustré sur son diagramme de classes à la Figure 4.3. La méthode *VerifierSig()* prend en argument l'objet signé de type *Message* ainsi que l'alias de l'agent signataire et renvoie un booléen pour dire si la vérification a réussi ou non, et si c'est le cas, récupère l'objet qui a été signé. Cette méthode utilise le même algorithme qui a servi à la signature et charge la clé publique à partir du certificat du signataire représenté par l'alias. Quant à la méthode *Decrypter()*, elle prend en argument un tableau d'octets et renvoie une chaîne de caractères correspondant au prix en clair. Cette dernière utilise aussi le même algorithme qui a été utilisé pour l'encryptage et décrypte le message en utilisant la clé publique de AS_{01} puisque les prix lui sont destinés. En bref, avec ces méthodes, AS_{01} récupère la liste des prix chiffrés et, pour chacun d'entre eux, vérifie d'abord la signature, récupère ensuite l'objet si la signature est correcte, puis décrypte l'objet et récupère finalement le prix en clair. À ce stade, ce dernier peut décider de la plate-forme dans laquelle l'achat devra se faire.

4.3 Tests et évaluation de l'implémentation

Nos tests ont été réalisés en utilisant 4 machines dont les principales caractéristiques sont décrites au Tableau 4.1. Cependant, dépendamment de la configuration utilisée, nous avons exécuté des agences différentes dans ces machines. La première configuration correspond au scénario optimiste et utilise une seule région qui contient un serveur de confiance *SC1* et trois agences. L'agence *Agence01* représente la plate-forme d'origine, et les agences *Agence11* et *Agence21* représentent les deux vendeurs. Quant à la deuxième configuration, elle correspond au scénario pessimiste et utilise deux serveurs de confiance, *SC1* et *SC2,* ainsi que trois agences. L'agence *Agence01* toujours comme plate-forme d'origine, et une agence dans chaque région représentant les deux vendeurs. L'agence *Agence11* représente le vendeur 1 dans la région 1, et *Agence12* représente le vendeur 1 dans la région 2.

Ainsi, la première machine exécute l'agence *Agence01* dans les deux scénarios. La deuxième exécute l'agence *Agence11* dans le premier scénario, et *Agence11* et *SC2* dans le deuxième. Quant à la troisième machine, elle exécute *Agence21* dans le premier scénario et *Agence12* dans le deuxième. Finalement, la dernière machine exécute *SC1* dans les deux scénarios. Le réseau utilisé est un réseau local de type Ethernet 100 Mbps.

Tableau 4.1 Caractéristiques des machines utilisées pour les tests

Type	Nom de la plate-forme		RAM (Mo)	Processeur	Système d'exploitation
	Scénario 1	Scénario 2			
Ordinateur de bureau	*Agence01*	*Agence01*	512	P IV, 1.8 GHz	Windows 2000 pro
Ordinateur de bureau	*Agence11*	*Agence11 SC2*	512	P IV, 1.8 GHz	Windows 2000 pro
Ordinateur de bureau	*Agence21*	*Agence12*	512	P IV, 1.8 GHz	Windows 2000 pro
Ordinateur de bureau	*SC1*	*SC1*	512	P IV, 1.8 GHz	Windows 2000 pro

4.3.1 Tests de l'implémentation

Nous avons testé les différentes fonctionnalités de notre protocole liées à la détection des attaques de la part d'éventuelles plates-formes malicieuses. Pour ce faire, nous avons élaboré un plan d'expérience se basant sur des scénarios d'attaques établis à partir des attaques décrites et analysées lors de la spécification de notre protocole. Dans chaque type de scénario, nous avons forcé une ou les deux plates-formes à agir de façon malicieuse auprès de l'agent ou des serveurs de confiance. Le Tableau 4.2 présente notre plan de tests ainsi que les résultats obtenus pour chaque scénario testé.

Tableau 4.2 Plan de tests et résultats

Type d'attaque	Scénario d'attaque	Implémentation	Résultat
Mascarade	*AM* est envoyé au mauvais endroit	*A11* envoie *AM* à *A'12* au lieu de *A12*	Détectée par *AS1* lors des vérifications
	AM est remplacé par un clone	*A11* crée un clone et l'envoie au lieu de *AM*	
Déni de service	Mauvaise exécution du code de *AM*	*A11* exécute le code et modifie l'état résultant	Détectée par *AS1* lors des vérifications
	Tuer *AM* ou le terminer sans notification	*A11* efface *AM* et n'envoie pas de message à *SC1*	
	AM est gardé plus longtemps que prévu	*A11* introduit un délai supplémentaire	*AS1* ne détecte pas cette attaque
Espionnage	Accès non autorisé aux données résultantes	*A12* essaie de lire le prix proposé par *A11*	Impossible à cause de l'encryptage
Altération	Modification du code de *AM*	*A11* envoie à *A12* un autre agent avec l'entête de *AM*	Détectée par *AS2* lors des vérifications
	Modification de la signature sur l'état d'entrée	*A11* envoie $D_{A11}(A_{11}, S_{11})$ avec une fausse signature	Détectée par *AS1* lors des vérifications
	Modification de l'état d'entrée	*A11* envoie un mauvais état d'entrée à *A12*	Détectée par *AS2* lors des vérifications
	Modification des entrées I_i et de l'état de sortie	*A11* envoie un mauvais état de sortie à *SC1*	Détectée par *AS1* lors des vérifications
	Modification des données	*A12* modifie le prix de *A11*	Détectée par *A01* lors de l'extraction des prix

4.3.2 Évaluation de l'implémentation

Nous avons implémenté notre protocole en prenant comme application la recherche du meilleur prix d'un certain produit. Pour ce faire, notre agent *AM* visite deux plates-formes, s'exécute dans chacune d'elles puis revient dans sa plate-forme d'origine avec la liste des prix recueillis. Comme présenté auparavant, nous avons testé deux cas de figure de notre protocole. Dans le premier, nous avons considéré un scénario optimiste dans lequel les deux plates-formes visitées appartiennent à la même région et interagissent donc avec le même serveur de confiance, *SC1*. Dans le deuxième, nous avons considéré un scénario pessimiste dans lequel chacune des plates-formes visitées est isolée dans sa région et interagit avec son serveur de confiance : *SC1* pour l'une et *SC2* pour l'autre.

Les performances de l'implémentation de notre protocole ont été testées dans les deux cas de figure et comparées avec celles d'un agent simple, appelé *AgentSimple*. Ce dernier exécute la même tâche que notre agent mobile *AM* mais n'utilise pas les fonctionnalités de sécurité de notre protocole. Ces performances ont été mesurées en terme de trafic généré par notre protocole et de temps d'exécution de ce dernier. Ce choix de critères de performances est justifié par le fait que, d'une part, notre protocole génère un trafic additionnel dû à l'utilisation du clone et des échanges de messages et, d'autre part, il augmente le temps d'exécution par l'utilisation de fonctionnalités de prévention et de détection des attaques. Cet exemple d'application va nous permettre d'étudier l'impact de l'utilisation de notre protocole afin de sécuriser une application à base d'agents mobiles.

Analyse du trafic généré

Pour analyser le trafic généré par notre protocole, nous avons utilisé le logiciel *EtherPeek v4.5*, développé par la société *WildPackets.com*. Ce logiciel permet de capturer tous les paquets qui transitent dans le réseau au moyen d'outils de filtrage qui peuvent se faire sur les ports, les protocoles, et les adresses source et destination. De plus, il fournit des statistiques et des graphiques représentant la répartition du trafic

suivant des paramètres prédéfinis par l'utilisateur. Nous avons ainsi mesuré le nombre d'octets échangés entre chaque paire de plates-formes intervenant dans notre implémentation et distingué le trafic dû au déplacement de l'agent, du clone et des communications. Ceci a été fait pour l'agent simple ainsi que pour les deux scénarios de notre agent mobile.

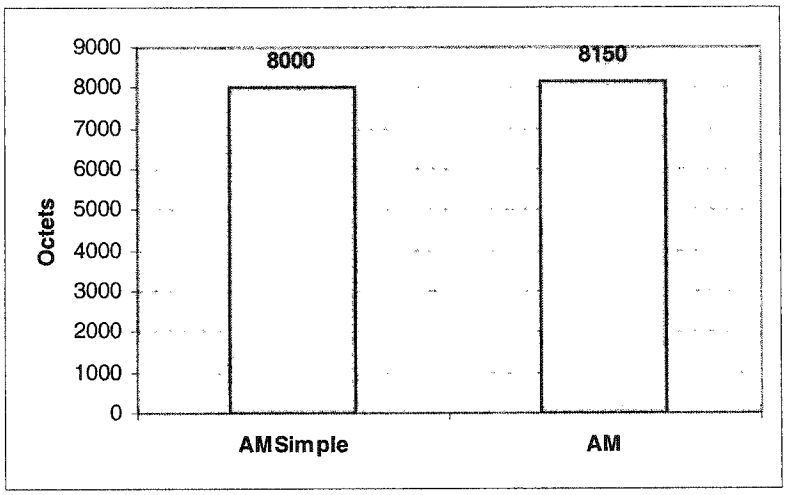


Figure 4.9 Comparaison des tailles des agents *AMSimple* et *AM*

La Figure 4.9 présente la taille des agents *AMSimple* et *AM*. En analysant cette figure, nous constatons que les tailles des deux agents sont sensiblement égales, *8 Ko* pour *AMSimple* et *8.15 Ko* pour *AM*. Ceci est justifié par le fait que les deux agents exécutent la même tâche, et les fonctionnalités de sécurité ainsi que les communications utilisées dans notre protocole sont effectuées par l'agent stationnaire de la plate-forme hôte. La petite différence de taille est engendrée par l'utilisation de quelques fonctions supplémentaires par *AM* pour manipuler les prix sécurisés qui ne sont pas utilisées par *AMSimple*.

Pour ce qui est de la répartition du trafic, nous avons utilisé le filtrage proposé par *EtherPeek* afin de mesurer exactement le trafic échangé entre chaque paire de plates-formes. Les Figures 4.10, 4.11 et 4.12 montrent respectivement la répartition du trafic engendré par *AMSimple* et par l'agent *AM* de notre protocole, noté *AMO* dans le scénario optimiste et *AMP* dans le scénario pessimiste.

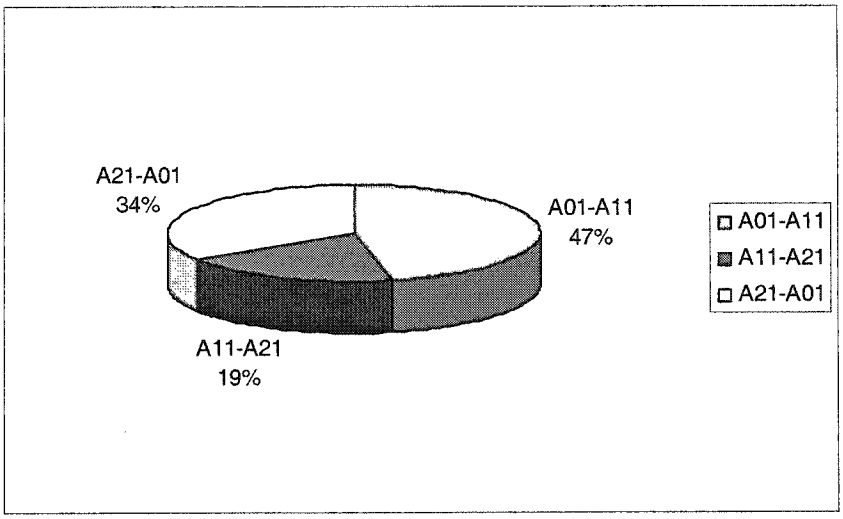


Figure 4.10 Répartition du trafic entre les plates-formes pour *AMSimple*

D'après la Figure 4.10, nous remarquons que le trafic généré entre la plate-forme d'origine *A01* et la première plate-forme de migration *A11* représente la plus grande partie du trafic (47 %), malgré le fait que c'est le même agent qui se déplace à chaque fois. Ceci est justifié par le fait que Grasshopper implémente la migration au sens faible dans laquelle le code nécessaire à l'exécution de l'agent est à chaque fois téléchargé à partir de la plate-forme d'origine. Ceci veut dire que quand *AM* se déplace de *A11* à *A21* le code est téléchargé à partir de *A01*, et seulement les résultats de l'exécution sont téléchargées à partir de *A11*. Cela veut dire aussi que quand l'agent revient dans sa plate-forme d'origine, seulement les résultats d'exécution sont migrés, étant donné que le code est déjà dans la plate-forme d'origine. De plus, dans les tests effectués, la région Grasshopper dans laquelle ont été enregistrées toutes les agences s'exécute dans la même machine que l'*Agence01*, engendrant ainsi un trafic supplémentaire. En effet, à chaque fois que des invocations ont été utilisées, la région a été contactée pour localiser l'agent invoqué. En réalité, ce trafic comporte trois portions : une due à la migration de l'agent, une autre aux informations échangées entre *A01* et la région Grasshopper, et la dernière due aux informations de signalisation échangées entre *A01* et *A11*.

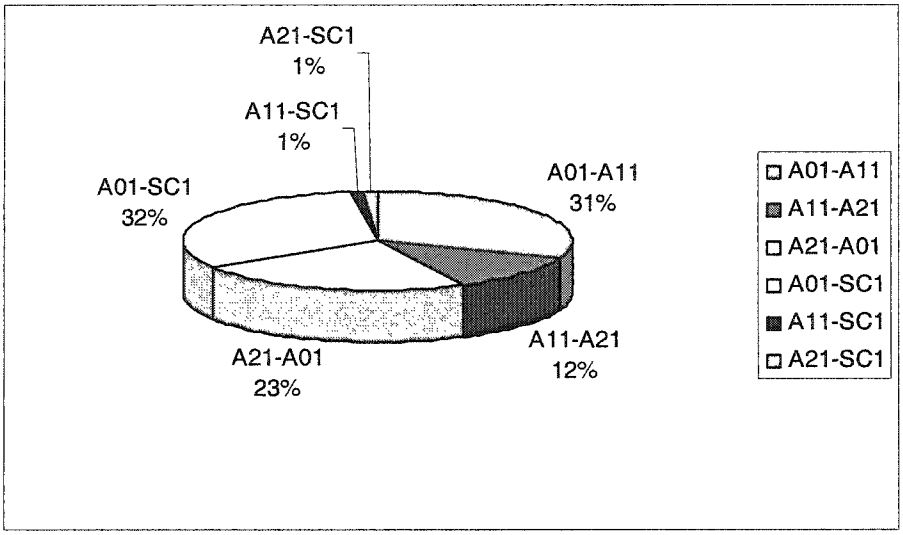


Figure 4.11 Répartition du trafic entre les plates-formes pour *AMO*

Dans le cas du scénario optimiste de notre protocole, la Figure 4.11 indique que la plus grande partie du trafic est celle générée entre *A01* et *SC1* (32%) et entre *A01* et *A11* (31%). Le premier trafic est généré suite à la migration du clone de *AMO,* de la plate-forme d'origine *A01* au serveur de confiance *SC1* et, le deuxième, suite à la migration de *AMO* de son origine *A01* au premier vendeur *A11*. Comme expliqué précédemment, ce trafic est composé d'une portion relative à la migration elle-même, une due aux échanges avec la région, et la dernière, à la signalisation entre les agences. Notons aussi que le trafic entre *A01* et *SC1* est légèrement plus grand car comprend la migration du clone et son retour à *A01*. Dans le retour, seulement un petit trafic est généré car le clone revient à sa plate-forme d'origine. Le trafic généré par les deux communications, entre *A11* et *SC1* d'une part et entre *A21* et *SC1* d'autre part, est le même mais ne représente qu'une petite fraction du trafic total (2%).

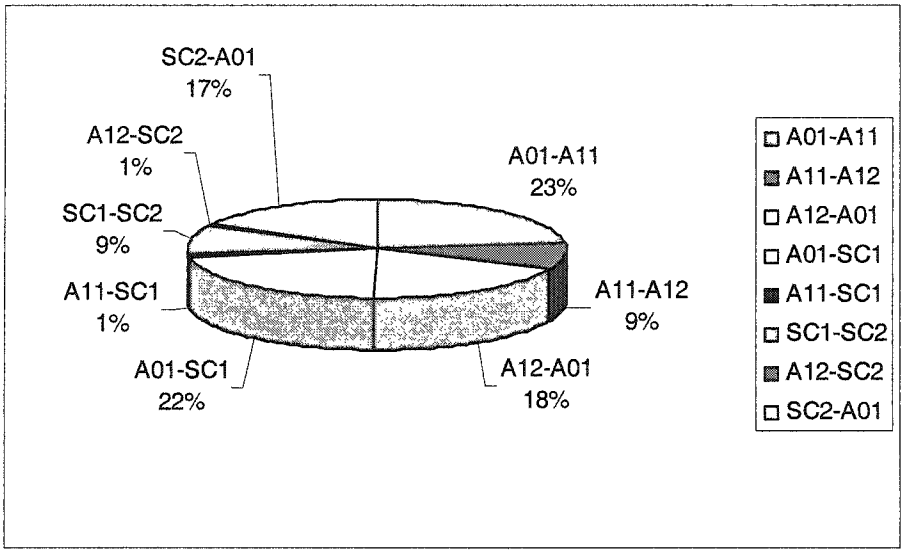


Figure 4.12 Répartition du trafic entre les plates-formes pour *AMP*

Pour ce qui est du scénario pessimiste de notre protocole, la Figure 4.12 confirme les résultats obtenus auparavant. En effet, la plus grande partie du trafic est générée entre *A01* et *A11* d'une part (23%), et entre *A01* et *SC1* (22%) d'autre part, et les communications constituent toujours une petite fraction du trafic (2%). Dans ce scénario, on a un trafic supplémentaire qui est généré par le déplacement du clone de *SC1* à *SC2*. Ce trafic représente le même pourcentage que celui généré par le déplacement de *AMP* de *A11* à *A12* (9%). Ceci s'explique par le fait que l'agent *AMP* et son clone ont la même taille, et le trafic est généré par la migration et par la signalisation entre les deux plates-formes dans les deux cas. Finalement, le pourcentage du trafic généré par le retour de l'agent *AMP* et de son clone à la plate-forme d'origine est sensiblement égal (respectivement 18% et 17%) et est constitué du trafic généré par la migration elle-même, d'un trafic de signalisation et d'une portion due à la communication avec la région.

Dans une autre mesure, nous nous sommes intéressés à distinguer la quantité de trafic généré par le déplacement de l'agent mobile lui-même, celui généré par le déplacement du clone et celui généré par les communications. La Figure 4.13 illustre

cette répartition pour le cas de *AMSimple*, le scénario optimiste avec *AMO* et pour le scénario pessimiste avec *AMP*.

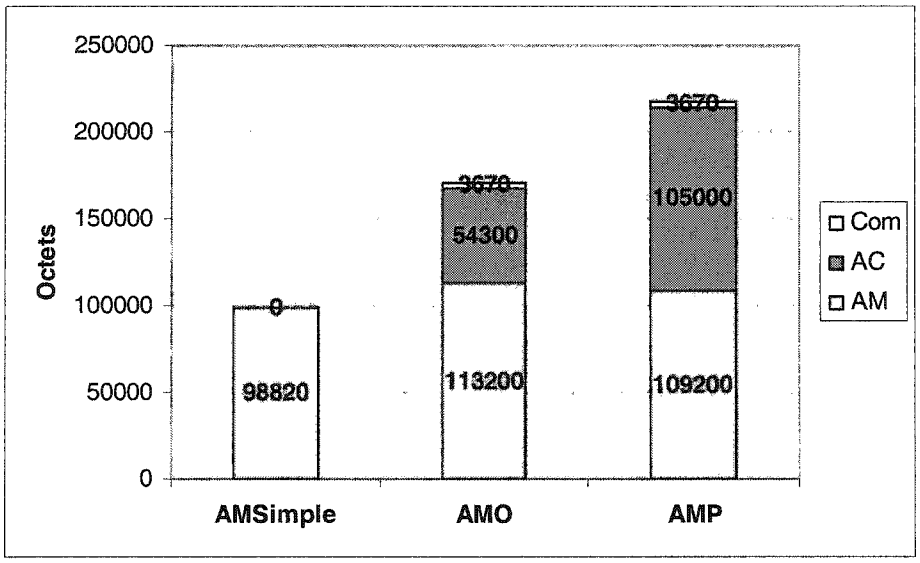


Figure 4.13 Trafic généré par *AM, AC* et les communications

Le trafic est totalement généré par les migrations de l'agent mobile dans le cas de *AMSimple,* alors qu'il est généré par l'agent mobile et son clone ainsi que par les communications dans les deux scénarios de notre protocole. Ceci est illustré par la Figure 4.13. Dans le scénario optimiste de notre protocole, le trafic généré par le clone représente à peu près la moitié du trafic généré par l'agent mobile lui-même. Ceci est dû au fait que l'agent mobile fait trois migrations, alors que le clone n'en fait que deux, dont une correspondant à son retour à la plate-forme d'origine, et engendre donc un petit trafic. Par contre, dans le scénario pessimiste, le trafic généré par le clone est quasiment égal à celui généré par l'agent mobile lui-même. Ceci est justifié par le fait que le clone se déplace à chaque déplacement de l'agent mobile, étant donné que, dans ce scénario, il y a une agence par région. Par ailleurs, le trafic généré par les communications est le même pour les deux scénarios, puisque les mêmes informations transitent dans le réseau dans les deux cas de figure.

D'après les résultats précédents, il apparaît que l'ajout des fonctionnalités de sécurité de notre protocole à une application augmente le trafic de 73% dans le scénario

optimiste et de 120% dans le pessimiste, ce qui donne une moyenne de 97%. Cela veut dire que notre protocole génère un trafic additionnel représentant moins du double de celui généré par une application non sécurisée. Cette quantité est raisonnable et constitue le prix à payer en trafic supplémentaire pour sécuriser une application à base d'agents mobiles.

Analyse du temps d'exécution

Le deuxième critère de performance que nous avons utilisé est le temps d'exécution de notre protocole ainsi que l'impact du nombre d'agents actifs sur ce dernier. Ce temps représente la durée totale d'exécution du protocole, depuis le départ de l'agent et de son clone de la plate-forme d'origine jusqu'à l'extraction des prix au retour de ces derniers. De même que pour l'analyse du trafic, nous avons mesuré le temps d'exécution de l'agent simple et des deux cas de figure de notre protocole afin de les comparer. Pour chaque cas, une dizaine de mesures ont été effectuées afin de diminuer les risques d'erreurs et la variabilité des mesures. Les temps d'exécution ont été mesurés par l'agent stationnaire de la plate-forme d'origine, *AgentStatique01*. Ce dernier prélève le temps système au départ des agents puis, après avoir récupéré les prix à leur retour, calcule la différence entre les deux pour avoir le temps d'exécution.

Avant d'exécuter un agent mobile, la machine virtuelle Java charge dans la mémoire cache les classes utilisées par ce dernier et les conservent. De ce fait, lors de la première mesure, ce temps constitue un temps additionnel par rapport aux mesures suivantes pour lesquelles les classes sont déjà chargées. Pour pallier cette différence notable entre la première mesure et les suivantes, nous ne considérons pas cette mesure. En fait, celle-ci sert justement à charger les classes nécessaires pour que, dans les mesures suivantes, seulement le temps d'instanciation des classes déjà chargées soit pris en compte. Cependant, nous avons quand même effectué plusieurs mesures dans chaque cas de figure et nous avons considéré leur moyenne.

Ainsi, nous avons comparé le temps global d'exécution de l'agent simple avec celui de l'exécution de notre protocole dans les deux scénarios, optimiste et pessimiste,

présentés précédemment. La Figure 4.14 illustre cette comparaison. Il en ressort que le temps d'exécution par rapport à l'agent simple augmente de 47% dans le cas optimiste de notre protocole et de 128% dans le cas pessimiste, ce qui donne une moyenne d'augmentation de 87%. Ceci reste raisonnable comme augmentation et constitue le prix à payer en temps d'exécution pour sécuriser une application utilisant des agents mobiles. Cette augmentation du temps d'exécution par rapport à l'agent simple est engendrée par les mécanismes d'encryptage et de signature numérique, les exécutions supplémentaires du clone et les communications effectuées. En effet, les prix résultant des négociations dans chaque plate-forme sont encryptés et signés numériquement, puis les opérations inverses sont réalisées lorsque l'agent revient à sa plate-forme d'origine, ce qui consomme du temps. De plus, du fait que l'exécution du clone se fait en décalage par rapport à l'agent mobile, ce dernier cumule du retard d'une vérification à une autre et engendre un temps additionnel, qui va se manifester avec l'arrivée en retard du clone, à la plate-forme d'origine, par rapport à l'agent mobile. Finalement, la dernière cause d'augmentation du temps d'exécution est engendrée par le temps d'établissement de la communication qui sert à envoyer les données d'exécution de l'agent au serveur de confiance dans chaque plate-forme visitée. Le fait que le temps d'exécution du cas pessimiste est beaucoup plus grand que pour le cas optimiste est justifié par le fait que, dans le premier cas, le clone migre autant de fois que l'agent engendrant ainsi un temps supplémentaire.

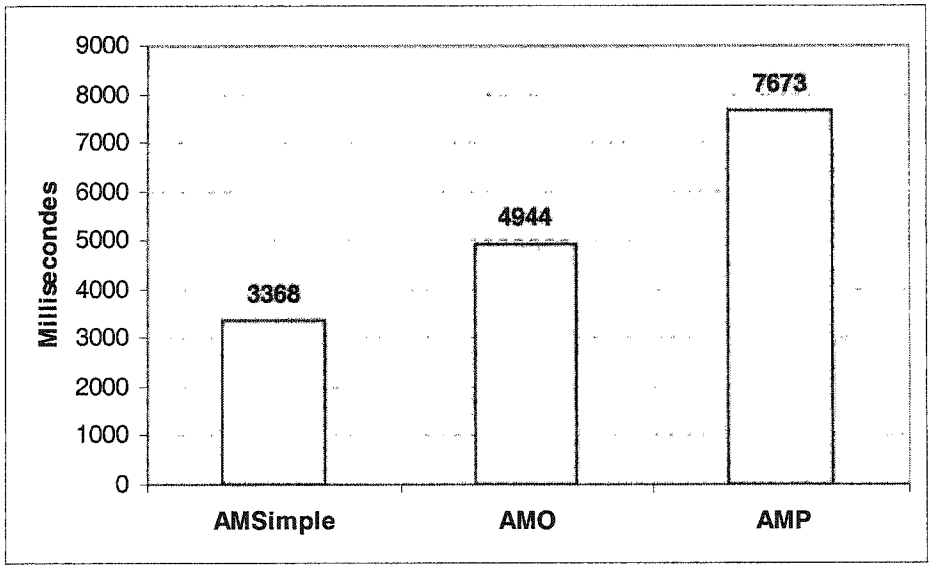


Figure 4.14 Comparaison des temps d'exécution de *AMSimple, AMO* et *AMP*

Dans une dernière mesure, nous avons étudié l'impact du nombre d'agents actifs sur le temps d'exécution de notre protocole. Pour ce faire, nous avons exécuté des agents supplémentaires dans chaque plate-forme intervenant dans notre protocole pour chaque scénario, et nous avons mesuré les temps d'exécution. Nous les avons ensuite comparés à ceux trouvés pour un agent simple. Les résultats trouvés sont illustrés à la Figure 4.15. Nous pouvons constater que les temps d'exécution des trois agents ainsi que leurs écarts augmentent avec l'augmentation du nombre d'agents actifs. Ceci est justifié par le fait que le temps CPU est partagé entre les différents agents s'exécutant, au lieu d'être dédié aux seuls agents de notre protocole. Ainsi, plus il y a d'agents plus le temps est partagé et plus la tâche de chacun d'eux prend plus de temps. Une autre raison est que, de part l'implémentation de Grasshopper, les agents ont la plus faible priorité comparée à celle des autres « *threads* » Java s'exécutant en même temps.

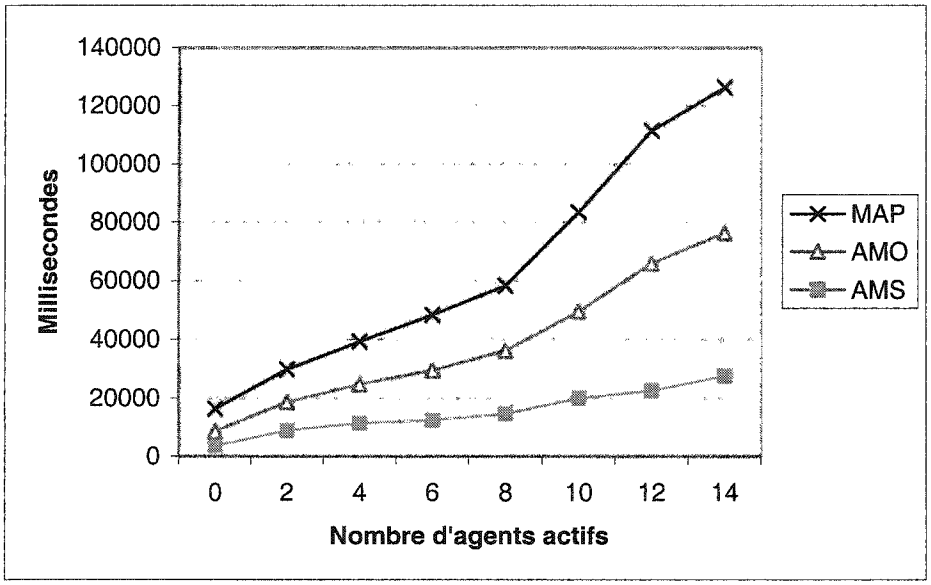


Figure 4.15 Impact du nombre d'agents actifs sur le temps d'exécution

Nous pouvons alors conclure que l'utilisation de notre protocole permet de sécuriser une application d'agents mobiles mais engendre un trafic et un temps d'exécution supplémentaires, qui constituent le prix à payer. Ce prix supplémentaire à payer par rapport à une application non sécurisée est en moyenne de 97% pour le trafic et de 87% pour le temps d'exécution.

CHAPITRE V
CONCLUSION

Dans ce chapitre, nous allons faire une synthèse de nos travaux concernant la protection de l'agent mobile contre les attaques de la part de plates-formes malicieuses. Nous présenterons par la suite leurs limitations théoriques et pratiques. Finalement, nous proposerons des indications en vue de recherches futures.

5.1 Synthèse des travaux

Dans le présent mémoire, nous avons présenté un protocole de sécurité permettant de protéger un agent mobile contre les attaques de plates-formes malicieuses, en utilisant un clone de référence. Nous sommes partis de l'idée d'exécution de référence et nous avons voulu effectuer la vérification en temps réel, tout en gardant la proximité géographique entre l'exécution de l'agent et sa vérification. Ainsi, nous sommes arrivés à l'utilisation de plusieurs serveurs de confiance qui peuvent vérifier l'exécution d'un agent mobile en exécutant son clone en parallèle.

Afin de concevoir notre protocole, nous avons tout d'abord recensé les types d'attaques auxquelles un agent mobile était sujet et la façon dont ces dernières étaient prévenues ou détectées avec les méthodes existantes dans la littérature. Nous avons par la suite étudié plus en détail les approches les plus efficaces à détecter les attaques les plus importantes. Finalement, à partir des faiblesses des unes et des forces des autres, nous avons conçu notre protocole en essayant de détecter le plus d'attaques possibles, sans compromettre l'avantage que procure le concept des agents mobiles par rapport au client/serveur.

Ainsi, nous avons utilisé le concept d'exécution de référence comme moyen de vérification de la bonne exécution du code de l'agent. Ces exécutions de référence sont effectuées en exécutant, en parallèle de l'exécution de l'agent mobile, un clone de ce dernier dans un serveur de confiance. Pour ce faire, nous avons supposé que le réseau était décomposé en régions et, dans chaque région, nous avons supposé l'existence d'un

serveur de confiance. Par la suite, nous avons spécifié notre protocole en décrivant les entités intervenantes ainsi que les tâches qu'elles doivent effectuer dans chaque étape. Finalement, nous avons vérifié, de façon théorique, comment notre protocole permettait de détecter les différentes attaques.

Après cette étape de spécification, nous avons choisi la plate-forme Grasshopper pour implémenter un prototype de notre protocole, en utilisant un réseau local de type ethernet. Avec le souci de choisir une application proche de la réalité, nous avons pris comme exemple un agent qui se déplace d'une plate-forme à une autre à la recherche du meilleur prix d'un certain produit. En plus de l'implémentation d'un agent simple représentant l'application non sécurisée, nous avons implémenté deux scénarios de notre protocole en restant les plus fidèles possibles à sa spécification. Le premier scénario est optimiste et suppose que toutes les plates-formes visitées par l'agent mobile appartiennent à la même région du réseau, et nécessite donc l'exécution d'un seul serveur de confiance. Le deuxième scénario est, quant à lui, pessimiste et suppose que toutes les plates-formes visitées sont dans des régions différentes; il utilise par conséquent un serveur de confiance pour chaque plate-forme visitée.

Finalement, nous avons implémenté notre protocole et nous avons démontré sa capacité à détecter les attaques voulues. De plus, nous avons testé les deux scénarios de notre implémentation et comparé leurs performances à ceux de l'agent mobile simple, i.e. celui de l'application non sécurisée. Les critères de performances qui ont servi de base pour la comparaison sont le trafic généré et le temps total d'exécution. Ces tests ont montré que, pour avoir une application sécuritaire, il fallait payer un certain prix relatif à la performance. Pour notre protocole, ce prix se chiffre à une moyenne de 97% plus de trafic et 87% plus de temps qu'une application non sécurisée. Ces chiffres restent raisonnables étant donné la solution de sécurité que ce protocole apporte à une application du commerce électronique à base d'agents mobiles.

Plus précisément, notre protocole permet d'assurer l'intégrité du code de l'agent, de son exécution, de ses données et de son itinéraire. En plus de sa capacité à identifier les plates-formes attaquantes, ce protocole est robuste, tolérant aux fautes et ne

compromet pas, dans une certaine mesure, l'autonomie de l'agent mobile et ses avantages par rapport au client/serveur.

5.2 Limitations des travaux

Bien que notre protocole permet de détecter plusieurs attaques contre un agent mobile, ce dernier connaît quand même des limitations reliées à la complexité de quelques attaques, à la limitation des ressources et du temps alloués.

Ainsi, notre protocole ne permet pas de détecter si la plate-forme n'accorde pas les ressources voulues à l'agent, le garde plus longtemps que prévu, ou encore, si elle introduit volontairement des délais inacceptables. En effet, dans ce cas, le serveur de confiance ne peut pas savoir la cause exacte de ce délai, qui peut très bien être normal si les ressources de la plate-forme sont occupées. Ce type d'attaque est très difficile à détecter et il n'existe pas à notre connaissance de mécanismes permettant de les prévenir à 100%. Pourtant, de telles attaques sont dangereuses pour des applications où le temps d'exécution est critique, comme pour l'achat d'actions boursières.

Notre implémentation a été réalisée en utilisant la plate-forme Grasshopper qui implémente la faible migration, et ne permet pas donc de tester effectivement l'efficacité de notre protocole à protéger l'état d'exécution. En effet, nous avons simulé l'état d'exécution par une simple variable d'état, ce qui ne correspond pas exactement à la spécification de notre protocole.

D'autres limitations sont dues aux difficultés à simuler les vraies attaques. En effet, certaines attaques sont impossibles à simuler sans modifier la façon dont Grasshopper exécute les agents. Les simulations de certaines attaques ont été alors simplifiées, soit en modifiant le code ou le résultat de son exécution. De ce fait, ces simulations ne reflètent pas les vraies attaques telles que des plates-formes malicieuses les feraient.

Finalement, une dernière limitation est due au fait que notre protocole n'a pas été testé avec une vraie application du commerce électronique et dans un environnement réel. En effet, malgré notre souci d'utiliser une application réaliste, cette dernière ne reflète pas forcément la réalité. De plus, nous avons utilisé un réseau local. De ce fait,

d'autres exigences ou des aléas pourraient survenir lors de l'utilisation de notre protocole dans un environnement réel.

5.3 Indications de recherches futures

D'après nos constats précédents, des travaux supplémentaires pourraient être envisagés afin de prouver les propriétés de notre protocole et d'améliorer ses performances. Ainsi, il serait intéressant de valider formellement notre protocole avec un logiciel tel que *SPIN* ou *SMV*. Une telle validation permettrait, en utilisant la logique temporelle linéaire (LTL), de vérifier qu'il n'y pas de blocages « deadlock » ou de divergences « livelock » dans tous les états du protocole et que ce dernier détecte bien les attaques prévues.

Afin de détecter les attaques relatives à l'introduction de délais abusivement, nous pouvons envisager d'utiliser des mécanismes supplémentaires. Nous pouvons par exemple demander aux plates-formes visitées d'envoyer au serveur de confiance une estimation du temps d'exécution de l'agent selon leurs disponibilités. Le serveur de confiance pourrait alors ignorer les résultats d'exécution dans une plate-forme aussitôt qu'il constate que cette dernière ne respecte pas le délai prévu. Par ailleurs, il serait intéressant de trouver un moyen d'exécuter les agents autrement que par invocation, tout en conservant la capacité de vérification des exécutions. Ceci permettrait d'améliorer les performances du protocole en diminuant le trafic. En effet, à chaque invocation, la région est contactée afin de localiser l'agent invoqué générant ainsi un trafic supplémentaire.

En outre, il serait intéressant de confirmer les résultats trouvés en implémentant notre protocole dans une application réelle. En effet, ceci permettrait de confronter son efficacité et sa capacité de détection dans un déploiement correspondant à la réalité et d'étudier sa robustesse. Ainsi, des ajustements pourraient être apportés afin d'adapter le protocole à l'environnement réel et de mesurer son coût effectif. De plus, le comportement de ce dernier pourrait être étudié, surtout face à l'exécution de plusieurs agents sécurisés par notre protocole en même temps.

BIBLIOGRAPHIE

[ALL01] G. Allée, *Sécurité des Agents Mobiles : Protocole d'Enregistrement d'Itinéraire par Agents Coopérants*, Mémoire de maîtrise, École Polytechnique de Montréal, Canada, Février 2001.

[CHE95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, G. Tsudik, "Itinerant Agents for Mobile Computing", *in IEEE Personal Communications*, vol. 2, n°. 5, Octobre 1995, pp. 34-49.

[COR99] A. Corradi, R. Montanari, C. Stefanelli, "Mobile agents integrity in e-commerce applications", *in Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*. Information Systems Vol. 24, No. 6, pp. 519-533, 1999

[FAR96] W. Farmer, J. Guttman, V. Swarup, "Security for Mobile Agents: Authentication and State Appraisal", *in Proceedings of the 4th European Symposium on Research in Computer Security* (ESORICS '96), Septembre 1996, pp. 118-130.

[FIP98] Foundation for Intelligent Physical Agents, "Agent Security Management", FIPA 98 Specification, Part 10, Version 1.0, Oct. 23, 1998. <URL: http://www.fipa.org/spec/fipa8a26.doc>.

[FRE96] A. Freier, P. Karlton, P. Kocher, *The SSL Protocol, Version 3.0*, Internet Draft, Mars 1996.
http://home.netscape.com/eng/ssl3/ssl-toc.html

[FUG98] A. Fuggetta, G.P. Picco, G. Vigna, "Understanding Code Mobility", *in IEEE Transactions on Software Engineering*, 24(5), May 1998.

[FUN99] S. Fünfrocken, "Protecting Mobile Web-Commerce Agents with Smartcards", *in Proceedings of the First International Symposium on Agent Systems and Applications / Third International Symposium on Mobile Agents* (ASA/MA'99), IEEE Computer Society, pp. 90-102, 1999.

[GIL95] D. Gilbert; M. Aparicio; B.Atkinson; S.Brady.; J.Ciccarino; B. Grosof, P. O'Connor; D. Osisek; S. Pritko; R. Spagna, L.Wilson, *IBM Intelligent Agent Strategy*, IBM Corporation, 1995.

[GRA01] Grasshopper, Release 2.2, *Basics and Concepts (Revision 1.0)*, Online Documentation, Mars 2001.
http://jce.iaik.tugraz.at/products/01_jce/documentation/index.php

[GRA96] R. S. Gray, "Agent Tcl: A Flexible and Secure Mobile-Agent System" *in Proceedings of the Fourth Annual Tcl/Tck Workshop* (TCL 96), pp. 9-23, Juillet 1996.

[HAS01] V. Hassler, *Security Fundamentals for e-commerce*, ARTECH HOUSE, INC., Massachusetts, 387 pages, 2001

[HOH00] F. Hohl, "A framework to Protect Mobile Agents by Using References States" *in Proceedings of ICDCS 2000*, 2000.
<URL:ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2000-03/TR-2000-03.ps.gz>

[HOH98] F. Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents fromt Malicious Hosts" *in Mobile Agents and Security*, pp. 92-113, G. Vigna (ed.), LNCS 1419, Berlin: Springer-Verlag, 1998.

[IAI02] Institute for Applied Information Processing and Communications, *Javadoc for IAIK-JCE 3.01*, Online Documentation, 2002. http://jce.iaik.tugraz.at/products/01_jce/documentation/index.php

[ITU93] ITU-T. *Information Technology – Open Systems Interconnection – The directory : Authentification FrameWork.* ITU-T Recommandation X.509, Novembre 1993

[KAR97] G. Karjoth, B.D. Lange, M. Oshima, "A security Model For Aglets" *in IEEE Internet Computing*, Août 1997, pp. 68-77.

[KAR98] G. Karjoth, N. Asokan, C. Gülcü, "Protecting The computation Results of Free Roaming Agents" *in Mobile Agents* (MA '98), pp. 195-207, K. Rothermel and F.Hohl (eds.), LNCS 1477, Springer-Verlag, 1998.

[KAR98b] N. Karnik, *Security in Mobile Agent Systems*, PhD. Dissertation, Departement of Computer Science, University of Minnesota, Octobre 1998.

[NEC96] G. Necula, P. Lee, "Safe Kernel Extensions Without Run-Time Checking" *in Proceedings of the 2nd Symposium on Operating System Design and Implementation* (OSDI '96), Seattle, Washington, Octobre 1996, pp. 229-243.
<URL: http://www.cs.cmu.edu/~necula/papers.html>.

[OAK01] S. Oaks, *Java Security, Second Edition,* O'REILLY, Californie, 599 pages, Mai 2001.

[OBJ00] Object Management Group, *OMG Agent Working Group,* Janvier 2000. <URL: http://www.objs.com/isig/agents.html>.

[OBJ97] Object Management Group, *Mobile Agent System Interoperability Facilities Specification,* OMG TC Document orbos/97-10-05, 1997. <URL: http://www.omg.org/cgi-bin/doc?orbos/97-10-05>.

[OBJ98] Object Management Group, *The Common Object request Broker: Architecture and Specification,* Revision 2.2, Feb. 1998.

[ORD96] J. Ordille, "When Agents Roam, Who Can You Trust?" *in Proceedings of the First Conference on Emerging Technologies and Applications in Communications*, Portland, Oregon, Mai 1996, pp. 188-191.

[PAP98] T. Papaioannou, J. Edwards, " Mobile agent technology in support of sales order processing in the virtual enterprise", *in Proceedings on 3rd IEEE/IFIP International Conference on Information Technology for Balanced Information Systems in Manufacturing,* Prague, Czech Republic, pp. 275-288, Kluwer Academic Publisher, 1998

[PFL96] C. P. Pfleeger, *Security in computing, second edition*, Prentice Hall, 1996, pp. 91-93.

[RIO98] J. Riordan, B. Schneier, "Environment Key Generation Towards Clueless Agents" *in Mobile Agents and Security*, pp. 15-24, G. Vigna (ed.), LNCS 1419, Berlin: Springer-Verlag, 1998.
<URL: http://www.counterpane.com/clueless-agents.html>.

[ROT99] V. Roth, "Mutual Protection of Co-operating Agents" *in Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pp. 275-285, J. Vitek and C. D. Jensen (eds.), LNCS 1603, Berlin: Springer-Verlag, 1999.

[SAN98] T. Sander, C. F. Tschudin, "Protecting Mobile Agents against Malicious Hosts" *in Mobile Agents and Security*, pp. 44-60, G. Vigna (ed.), LNCS 1419, Berlin: Springer-Verlag, 1998.

[SCH96] B. Schneider, *Applied Cryptography – Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, 2nd edition, 1996

[SCH97] F. B. Schneinder, "Towards Fault-tolerant and secure agentry" *in Proceedings of the 11th International Workshop on Distributed Algorithms*, Saarbrücken, Germany: Septembre 1997.
<URL:http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Summarize/ncstrl.cornell/TR97-1636>.

[SOH98] S. Sohn, K. J. Yoo, "An architecture of electronic market applying mobile agent technology", in *Proceedings of 3rd IEEE Symposium on Computers and Communications*, Athens, Greece, pp. 359-364, 1998.

[TAR96] J. Tardo, L. Valente, "Mobile Agent Security and Telescript" *in Proceedings of IEEE* (COMPCON '96), Santa Clara, California, pp. 58-63, Février 1996.

[VAR00] V. Varadharajan, "Security Enhanced Mobile Agents" *in Proceedings of the 7th ACM conference on Computer and communications security*, Novembre 2000, pp. 200-209.

[VIG97] G. Vigna, "Protecting Mobile Agents Through Tracing" *in Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems*, Jyvälskylä, Finland, Juin 1997.
<URL: http://www.cs.ucsb.edu/~vigna/listpub.html>.

[VIG98] G. Vigna (ed.), "Mobile agent and security", in *Lecture Notes in Computer Science,* Spring-Verlag, 1419(12), 1998.

[VIG98b] G. Vigna, "Cryptographic Traces for Mobile Agents, in *Mobile agents and security*, G. Vigna (ed.), Springer-Verlag, 1998, pp. 137-153.

[WAH93] R. Wahbe, S. Lucco, T. Anderson, "Efficient Software-Based Fault Isolation" *in Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, ACM SIGOPS Operating Systems Review, pp. 203-216, Décembre 1993.

[WIL99] U. G. Wilhelm, *A technical Approach to Privacy based on Mobile Agents Protected by Tamper-Resistant Hardware,* PhD. dissertation, No. 1961, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1999.

[YEE99] B.S. Yee, "A sanctuary for Mobile Agents" *in Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pp. 261-273, J. Vitek and C.D Jensen (eds.), LNCS 1603, Berlin: Springer-Verlag, 1999.

<URL: http://www-cse.ucsd.edu/users/bsy/pub/sanctuary.pdf>.